Aplicación de metaheurísticas para el diseño y planificación de rutas turísticas en destino usando Python

Oriol Moner Lasheras

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: Ricardo López Ruiz
4 de septiembre de 2023

Resumen

El turismo siempre ha exigido una planificación rigurosa para visitar el máximo número de puntos de interés (PDIs) posibles una vez llegado a un destino con un tiempo limitado. Sería de gran interés tener un modelo que facilitara la tarea de la planificación de las rutas turísticas y dejara para el turista únicamente el problema de ponderar los PDIs según la relevancia que el le quiere dar a cada uno. Tenemos un problema de optimización en el que tenemos que maximizar la cantidad y calidad de los PDIs a visitar con una restricción de tiempo.

En el capítulo 1 veremos problemas similares que han sido formulados y solucionados previamente, como es el Travel Salesman Problem, el primero de esta índole, o el Team Orienteering Problem (TOP), en el que nos fijaremos ya que es el que más representa el problema que queremos abordar.

Sin embargo al orientar el TOP al turismo es llamado Tourist Trip Design Problem (TTDP) ya que queremos aplicarle dos restricciones adicionales, una de ventanas de tiempo para las visitas de los PDIs y otra de categorización de los PDIs para restringir a un máximo y/o a un mínimo los PDIs de una determinada categoría por día.

Seguidamente haremos una formulación del problema de optimización lineal entera mixta que surge de nuestro planteamiento. Para ello iremos de menos a más sumando complejidad al problema, empezamos formulando el Orienteering Problem, que es el TOP para solo una ruta, luego formulamos el TOP y por último añadiremos las restricciones dando lugar a nuestro TTDP.

En el capítulo 2 tras ver la formulación del problema de optimización vemos que adquiere una complejidad muy elevada para hallar una solución exacta, en especial cuando aumenta la cantidad de PDIs, esto nos hace poner el foco en métodos de solución aproximada como son los métodos de resolución metaheurísticos.

Tras ver algunos de ellos, nos centraremos en el método Greedy Randomized Adaptative Search Procedure (GRASP) para usarlo en nuestro problema. Este método tiene una fase de construcción, en la que añadimos uno a uno los PDIs según una función objetivo y una parte aleatoria, y una fase de optimización que intentara mejorar la ruta ya construida, para esta fase usaremos un método de búsqueda local llamado Variable Neighborhood Descent (VND) que busca el mínimo local del vecindario de la ruta.

En el capítulo 3 veremos como los algoritmos del GRASP serán implementados en Python, donde se hará una generación de datos y unas posteriores pruebas con los diferentes parámetros buscando con que combinación de ellos el algoritmo presenta mejores resultados. El código puede encontrarse en el Anexo I.

Summary

Tourism has always demanded a rigorous planning to visit the maximum number of points of interest (POIs) possible once arrived at a destination with a limited time. It would be of great interest to have an algorithm that would solve the task of planning tourist routes and leave for the tourist only the problem of weighting the POIs according to the relevance he wants to give to each one. We have an optimization problem where we have to maximize the quantity and quality of the POIs that the tourist will visit with a time constraint.

In chapter 1 we will see similar problems that have been formulated and solved previously, such as the Travel Salesman Problem, the first of its kind, or the Team Orienteering Problem, which is the one that best represents the problem we want to address.

However, when oriented to tourism it is called Tourist Trip Design Problem (TTDP) since we want to apply two additional restrictions, one of time windows for the visits of the POIs and another one of categorization of the POIs to restrict to a maximum and/or a minimum the POIs of a certain category per day.

Next we will formulate the mixed integer linear optimization problem that arises from our approach. To do this we will be adding complexity gradually to the problem, we will start formulating the Orienteering Problem, which is the TOP for only one route, then we will formulate the TOP and finally we will add the constraints giving rise to our TTDP.

In chapter 2 after seeing the formulation of the optimization problem we see that it acquires a very high complexity to find an exact solution, especially when the number of POIs increases, this makes us look at approximate solution methods such as metaheuristics resolution methods.

After seeing some of them, we will focus on the Greedy Randomized Adaptive Search Procedure (GRASP) that will be used in our problem. This method starts with a construction phase in which we add one by one the POIs according to an objective function and a random step, then an optimization phase takes place to try to improve the route already constructed, for this phase we will use a local search method called Variable Neighborhood Descent (VND) that looks for the local minimum of the neighborhood of the route.

In chapter 3 we will see how GRASP algorithms will be implemented in Python, where we will generate data to test the algorithm performance while we will be changing the parameters of the algorithm to find the best combination of them. The code can be found in Anexo I.

Índice general

Capítulo 1

Introducción

1.1. Motivación

La industria turística es una de las más grandes en el mundo, en 2019 antes de la crisis provocada por la pandemia el turismo obtenía aproximadamente un 10 % del PIB mundial mientras que en regiones como Canarias es el principal sector industrial con un 35 % del PIB en este caso. A pesar de ser un sector tan amplio todos sus clientes están sujetos, en mayor o menor medida, a la planificación del viaje y la toma de decisiones dentro de él.

Nos fijamos en que a la hora de preparar unas vacaciones las personas tienen un mar de posibilidades en todos los frentes, destino, transporte, alojamiento, lugares dentro de su destino... etc. ¿Cómo podemos ayudar en la toma de decisiones? La mayoría de ellas dependen del gusto del viajante pero hay algunos factores que en su elección solo se busca la optimización de recursos como tiempo o dinero, esos son los factores con los que podríamos trabajar. Nosotros nos vamos a centrar en la planificación de rutas en destino, es decir, de la planificación de las rutas que nos llevaran a los puntos de interés de la zona que estamos visitando.

Vemos la necesidad de plantear un problema que tenga por input los puntos de interés con sus respectivas distancias entre ellos y por output la ruta a seguir para visitarlos, ya que cuando el número de puntos avanza para el ser humano es imposible saber cual es la mejor ruta y un quebradero de cabeza visualizar una ruta verdaderamente eficiente. Los beneficios que esto traería son claros, personalización de las recomendaciones como se expone en [1], ahorro en tiempo y dinero de los viajantes a través de la reducción del kilometraje total del viaje y soluciones a problemas como cumplir las ventanas de tiempo de las visitas o alcanzar un numero minimo o maximo de tipos de puntos de interés en un día o viaje.

1.2. Planteamiento

Procedemos a plantear el problema, como hemos dicho nos vamos a centrar en el itinerario de la ruta del turista, así que vamos a crear un escenario a partir del cual ya podemos entrar en la formulación matemática, destacamos una serie de puntos:

- Comenzamos situando a nuestro turista en su ya elegido lugar de destino donde cuenta con numerosos puntos de interés, muchos más de los que podría visitar en el tiempo estipulado que tiene para pasar sus vacaciones, debe gestionar sus rutas para visitar el máximo número de ellos.

- Tendremos en cuenta el lugar del hospedaje del turista, el cual supondremos que es único para toda su estancia, este será el punto de partida y llegada de cada ruta que diseñemos, será nuestro punto de interés número 0.

- Para diseñar la ruta tendremos que saber cuales son los PDIs (puntos de interés, usaremos su acrónimo) de la zona de los cuales no nos interesara saber su ubicacion exacta sino tan solo la distancia que se llevan unos con otros, sus distancias serán valores únicos y constantes durante todo el problema y todos tienen una distancia asociada entre sí.
- Cada PDI tendrá una duración de visita asociada.
- También debemos conocer de los PDIs sus ponderaciones, ya que el turista puede estipular cuánto valoraría visitar un PDI u otro, entonces ya no se trata de maximizar el número de PDIs visitados sino de maximizar la función que se crea al sumar las ponderaciones de los PDIs.
- Contamos con que el turista va a estar uno o varios días en sus vacaciones, también con que todos los días va a tener el mismo tiempo para visitar los PDIs.
- Representando los horarios de los PDIs vamos a estipular que cada PDI tenga una ventana de tiempo, es decir debemos llegar posteriormente a su inicio y terminar la visita antes de su cierre.
- Cada PDI va a pertenecer a una categoría, el turista podrá establecer un mínimo y/o máximo de visitas a cada categoría de PDIs por día, esto representaría poder exigirle a la solución que visite exactamente dos restaurantes por día o un máximo de una playa al día por ejemplo.

Con este planteamiento debemos crear un problema de optimización que se ajuste a los puntos expuestos, para ello vamos a hacer una revisión de modelos ya estudiados que se ajustan a nuestros requerimientos.

1.3. Modelos previos

Remontándonos a cuál podría ser el primer problema formulado de este estilo encontramos el Travel Salesman Problem(TSP) formulado por Karl Menger en 1930, como se describe en [2], el cual trata de un vendedor ambulante que tiene un conjunto de ciudades que visitar una única vez con unas distancias entre cada una de ellas determinadas, tras visitarlas todas vuelve al lugar desde donde partió, se plantea entonces la ruta mas rapida. A continuación de este problema surge el Vehicle Routing Problem(VRP) que se describe de forma similar pero contamos con varios vendedores, tenemos entonces que crear una ruta para cada uno de ellos.

Sin embargo, ambos problemas se diferencian del nuestro en algo clave, tanto en el TSP como en el VRP tenemos un conjunto de ciudades que visitar y un tiempo que reducir, en cambio, en nuestro planteamiento tenemos un tiempo fijo que gastar y un número de ciudades a visitar que aumentar.

Pasamos a considerar entonces el Orienteering Problem(OP) definido por Tsiligirides en 1984 [3], el cual está inspirado en el deporte de orientación y ahora si tenemos un tiempo dado y una cantidad de puntos a visitar que maximizar. Este modelo sería muy similar, salvo que en el OP el último punto visitado puede no ser el origen, además solo nos valdría para el caso en el que el turista solo fuera a hacer una ruta y nosotros queremos considerar más días.

Nos fijamos entonces en el Team Orienteering Problem(TOP), descrito en [4] que a diferencia del OP este tiene más personas que pueden visitar los puntos y debemos de planificar una ruta por cada jugador, para nosotros la cantidad de jugadores va a representar la cantidad de días en los que podamos hacer una ruta por los PDIs. Vemos que el VRP es la variante multipersona del TSP como lo es el TOP de el OP.

A partir del TOP han surgido múltiples variantes al hacerle modificaciones, así es que una de ellas es el Tourist Trip Design Problem (TTDP), descrito en [5], problema donde vamos a focalizar todo el estudio. Entre el TOP y el TTDP no hay otra diferencia que el enfoque que se le da al problema, es decir,

va a considerarse como un TTDP cuando todas las restricciones y particularidades que le añadas al TOP se enfoquen para el caso del turismo. Por ejemplo el tiempo gastado durante la visita de un PDI puede ser considerado o no en el TOP pero siempre lo será en el TTDP.

Por supuesto al TTDP también se le puede añadir una infinidad de modificaciones, tantas como el turista pueda precisar, nosotros nos centraremos en dos, una son las ventanas de tiempo donde se tiene en cuenta la llegada y salida a los PDIs para el cumplimiento del horario de este y otra es la categorización de los PDIs, es decir que al formar categorías en los PDIs puedas restringir un máximo y un mínimo de una determinada categoría por dia.

1.4. Formulación

1.4.1. Problema OP

Una vez hecho el análisis de los modelos previos empezaremos a formular los problemas de menos a más. Comenzaremos por el OP, basándonos en [5] ya que se acerca más a nuestro problema que el TSP y es más sencillo que el TOP. Nos enfrentamos a un problema de optimización entera mixta en el que trataremos de maximizar una función objetivo con una serie de restricciones en las variables, comenzamos declarando las variables y constantes necesarias para la formulación explicando su representación y significado en nuestro problema particular del turista:

- Sea N el número de PDIs enumerados 1,2,...,N donde s = 1 = N es el nodo de salida y de llegada.
- p_i será el beneficio de visitar el PDI i.
- c_{IJ} será el tiempo gastado en viajar del PDI i al j
- V_I será el tiempo gastado en visitar el PDI i.
- T_{max} es el tiempo máximo a gastar en todo el día.
- Diremos que x_{ij} es igual a 1 si en algún punto de la ruta se ha ido del PDI i al j y 0 en el caso contrario, es decir, se ha gastado el tiempo c_{ij}.
- Denotamos por u_i al puesto que ocupa el PDI i en la ruta en caso de que haya sido visitado, en caso contrario 0.

Con estas variables montamos la formulación del problema:

$$\max \sum_{i=2}^{N-1} \sum_{j=2}^{N} p_i x_{ij} \tag{1.1}$$

$$t.q.$$

$$\sum_{j=2}^{N} x_{1j} = \sum_{i=1}^{N-1} x_{iN} = 1, \tag{1.2}$$

$$\sum_{i=1}^{N-1} x_{1r} = \sum_{j=2}^{N} x_{rj} \leq 1, \quad \forall r = 2,...,N-1, \tag{1.3}$$

$$\sum_{i=1}^{N-1} \sum_{j=2}^{N} (c_{ij} + v_i x_{ij}) * x_{ij} \leq T_{max}, \tag{1.4}$$

$$2 \leq u_i \leq N, \quad \forall i = 1,2,...,N, \tag{1.5}$$

$$u_i - u_j + 1 \leq (N-1)(1-x_{ij}), \quad \forall i,j = 2,...,N, \tag{1.6}$$

$$x_{ij} \in \{0,1\}, \quad \forall i,j = 1,...,N. \tag{1.7}$$

Vamos a explicar cada una de las restricciones vistas:

- (1.1) Antes de comenzar con las restricciones tenemos la función objetivo que sale de sumar el peso de cada PDIs visitado
- (1.2) Nos aseguramos de que la ruta empieza en el PDI 1 y termina en el *N*
- (1.3) Nos aseguramos de que el camino empezado en el nodo 1 y terminado en el *N* está conectado y cada nodo es visitado máximo una vez
- (1.4) Controla que no nos pasemos del tiempo límite del dia
- (1.5) y (1.6) Aseguran que no hay varios círculos, es decir que todos los PDIs visitados están en el mismo camino.
- (1.7) Es necesario por la definición que le hemos dado.

Tenemos pues la formulación del OP, que se asimila a nuestro problema con solo un día de viaje. Ahora que ya tenemos una base en la formulación nos resultará más fácil continuar hasta la formulación del TTDP. Pasamos a realizar modificaciones en las restricciones y función objetivo ya planteadas para sacar el TOP.

1.4.2. Problema TOP

Para construir la formulación del TOP tenemos que empezar modificando las variables anteriormente definidas, generalmente añadirle una dimensión más a los índices ya que ahora cualquier PDI puede ser visitado en más de un día.

Enunciamos todas de nuevo:

- Sea *N* el número de PDIs enumerados 1,2,...,*N* donde s = 1 = *N* es el nodo de salida y de llegada.
- Sea *k* el número de rutas a realizar, es decir el número de días en los que se visitarán los PDIs.
- p_i será el beneficio de visitar el PDI i.
- c_{ij} será el tiempo gastado en viajar del PDI i al j
- v_i será el tiempo gastado en visitar el PDI i.
- T_{max} es el tiempo máximo a gastar en todo el día.
- Diremos que x_{ijm} es igual a 1 si en algún punto de alguna ruta se ha ido del PDI i al j y 0 en el caso contrario.
- Diremos que y_{im} es igual a 1 si se ha visitado el PDI i en la ruta m y 0 en el caso contrario.
- Denotamos por u_{im} al puesto que ocupa el PDI i en la ruta m en caso de que haya sido visitado, en caso contrario 0.

Rescribimos la nueva formulación con las nuevas variables:

$$\max \sum_{m=1}^{k} \sum_{i=2}^{N-1} p_i y_{im} \tag{1.8}$$

$$t.q.$$

$$\sum_{m=1}^{k} \sum_{j=2}^{N} x_{1jm} = \sum_{m=1}^{N-1} \sum_{i=1}^{N-1} x_{iNm} = k, \tag{1.9}$$

$$\sum_{m=1}^{k} y_{rm} =\leq 1, \quad \forall r = 2,...,N-1, \tag{1.10}$$

$$\sum_{i=1}^{N-1} x_{irm} = \sum_{j=2}^{N} x_{irj} = y_{rm}, \quad \forall r = 2,...,N, \quad \forall m = 1,...,k \tag{1.11}$$

$$\sum_{i=1}^{N-1} \sum_{j=2}^{N} (c_{ij} + v_i x_{ij}) x_{ijm} \leq T_{max}, \quad \forall m = 1,..,k, \tag{1.12}$$

$$2 \leq u_{im} \leq N, \quad \forall i = 1,2,...,N, \quad \forall m = 1,...,k, \tag{1.13}$$

$$u_{im} - u_{jm} + 1 \leq (N-1)(1 - x_{ijm}), \quad \forall i,j = 2,...,N, \quad \forall m = 1,...,k, \tag{1.14}$$

$$x_{ijm}, y_{im} \in \{0,1\}, \quad \forall i,j = 1,...,N, \quad \forall m = 1,...,k \tag{1.15}$$

Vemos que la formulación es muy similar, tan solo se ha añadido una nueva restricción (1.11) y en la demás se puede ver su restricción gemela, igualmente explicamos la utilidad de cada una:

- (1.8) La función objetivo que sale de sumar el peso de cada PDIs visitado.
- (1.9) y (1.10) Nos aseguramos de que cada ruta empieza en el PDI 1, termina en el N y que cada punto que no es el 1 y el N es visitado tan solo una vez.
- (1.11) Nos aseguramos que un flujo de una unidad puede pasar a lo largo de cada ruta que conecta el PDIs 1 y el N, garantizando así que el camino está conectado
- (1.12) Controla que no nos pasemos del tiempo límite del dia
- (1.13) y (1.14) Aseguran que no hay varios círculos, es decir que todos los PDIs visitados están en el mismo camino.
- (1.15) Es necesario por la definición que le hemos dado.

Ya tenemos el TOP formulado correctamente, el último paso será añadir las restricciones orientadas a la planificación de una ruta turística para que tenga el carácter de un TTDP.

1.4.3. Problema TTDP

Como hemos dicho anteriormente vamos a añadir a nuestro problema la restricción de las ventanas de tiempo y la limitación de visitas a una categoría por día.

Empezaremos por las ventanas de tiempo, retomaremos la formulación anterior ya que será una evolución del problema anterior añadiendo nuevas restricciones, sin embargo sí que tenemos que añadir nuevas variables al problema:

- Sea $a_i \in [0, T_{max}]$ el tiempo de llegada al PDI i.
- b_i es la hora de apertura de la ventana de tiempo del PDI i, no podremos llegar a visitarlo antes de esa hora.
- e_i es la hora de cierre de la ventana de tiempo del PDI i, se tiene que haber llegado al PDI y terminado el tiempo de visita antes de esa hora.
- Sea G el número de categorías a las que los PDIs pueden pertenecer.
- Sea g_h el conjunto de los PDIs que pertenecen a la categoría h.
- k_h será el número mínimo de PDIs de la categoría h que tienen que ser visitados cada día.
- l_h será el número máximo de PDIs de la categoría h que pueden ser visitados cada día.
- M un número suficientemente grande, no representa ningún parámetro del problema tan solo es usado para formularlo.

Vemos que no hemos tomado en cuenta a qué día nos referimos en las nuevas variables ya que asumimos que todos los PDIs abren y cierran a la misma hora y las restricciones de categoría son iguales para todos los días. Una vez declaradas las nuevas variables añadimos estas 6 nuevas restricciones:

$$b_i Y_{im} \leq a_i, \quad \forall i = 1,...,N, \quad \forall m = 1,...,k, \tag{1.16}$$

$$a_i \leq e_i(Y_{im} + ((1 - Y_{im})M)), \quad \forall i, j \in= 1,...,N, \quad \forall m = 1,...,k, \tag{1.17}$$

$$a_i + c_{ij} + vi \leq a_j + M(1 - X_{im}), \quad \forall i, j = 1,...,N, i \neq j, \quad \forall m = 1,...,k, \tag{1.18}$$

$$a_j \leq a_i + e_{ij} + v_i + M(1 - X_{im}), \quad \forall i, j = 1,...,N, i \neq j, \quad \forall m = 1,...,k, \tag{1.19}$$

$$k_h \leq \sum_{i \in g_h} Y_{im} \quad \forall h, = 1,...,G, \quad \forall m = 1,...,k, \tag{1.20}$$

$$\sum_{i \in g_h} Y_{im} \leq l_h \quad \forall i, j = 1,...,N, \quad \forall m = 1,...,k, \tag{1.21}$$

Hemos añadido seis restricciones, las cuatro primeras corresponden a la restricción de las ventanas de tiempo y las otras dos a la restricción por categorías procedemos a explicarlas:

- (1.16) Restringimos que lleguemos tras la apertura del PDI.
- (1.17) Restringimos que terminemos la visita del PDI antes de que cierre.

- Las restricciones (1.18) y (1.19) garantizan que el PDI j es visitado justo después del PDI i, es decir, que los tiempos están perfectamente enlazados.
- (1.20) Controlamos que el número de visitas a los PDIs de todas las categorías llegue al mínimo diario.
- (1.21) Controlamos que el número de visitas a los PDIs de todas las categorías no supere el máximo diario.

Ahora si tenemos finalmente el problema formulado con las dos restricciones que le queríamos añadir. Vamos a focalizarnos en encontrar un método de solución, para este tipo de problemas existen dos variantes: métodos exactos, donde encontraremos la solución óptima y métodos aproximados, donde encontraremos un solución con buenos resultados pero no podemos asegurar que es la óptima o cuanto de cerca esta de ella.

Vemos que en nuestro caso el problema ha ido escalando de tamaño tanto en variables como en restricciones, esto hace que los métodos exactos empiecen a no poder garantizar una solución óptima cuando consideramos conjuntos de PDIs suficientemente grandes.

Debemos buscar métodos de solución aproximada como son los métodos metaheurísticos, los cuales a pesar de la alta complejidad del problema nos van a permitir obtener una solución aproximada con la que nuestro viajero puede sacar el máximo beneficio a sus vacaciones, vemos en [6] como la gran mayoria de metodos usados para solucionar el TTDP son aproximados.

Capítulo 2

Metaheurísticas

En este capítulo vamos a realizar una revisión bibliográfica de los métodos metaheurísticos tras la cual desarrollaremos en profundidad el método empleado para la resolución del problema antes formulado.

2.1. Introducción

Las herramientas de optimización han mejorado mucho en las dos últimas décadas. Esto se debe a varios factores: (1) avances en la teoría de la programación matemática y el diseño algorítmico; (2) la rápida mejora del rendimiento de los ordenadores; (3) una mejor comunicación de las nuevas ideas y su integración en programas informáticos complejos de uso generalizado. En consecuencia muchos problemas que durante mucho tiempo se consideraban inalcanzables se resuelven actualmente, a veces en tiempos de cálculo muy moderados.

Este éxito, sin embargo, ha llevado a investigadores y profesionales a abordar instancias mucho mayores y clases de problemas más difíciles. Muchos de ellos sólo pueden resolverse de forma heurística. Por ello, cada año aparecen miles de artículos que describen, evalúan y comparan nuevas heurísticas. Estar al día de tanta literatura es todo un reto. Para organizar el estudio de las heurísticas se necesitan metaheurísticas o marcos generales de construcción de heurísticas.

Según expone Francisco Herrera en [7] una definición de la metaheurística sería una familia de algoritmos aproximados de propósito general, que suelen ser procedimientos iterativos que guían una heurística subordinada de búsqueda, combinando de forma inteligente distintos conceptos para explorar y explotar adecuadamente el espacio de búsqueda.

En comparación con otros métodos de solución podemos destacar en su favor: son algoritmos de propósito general, tienen gran éxito en la práctica y son fácilmente implementables y paralelizables.

En cambio en su contra podemos destacar: no aseguran la solución óptima ya que son métodos aproximados, no son determinísticos sino probabilísticos y no siempre tiene una base teórica estable.

Unos de los factores que hacen interesante su uso principalmente son:

- Cuando no tenemos un método exacto de resolución, o éste requiere mucho tiempo de cálculo y memoria
- Cuando no se necesita la solución óptima, basta con una de buena calidad

Los diferentes tipos de metaheurísticas vienen de ámbitos muy particulares y diferentes entre sí, sin embargo hablando de su funcionamiento podemos destacar que para obtener buenas soluciones, cualquier

algoritmo de búsqueda debe establecer un balance adecuado entre dos características contrapuestas del proceso:

- Intensificación, es la cantidad de esfuerzo empleado en la búsqueda de la región actual (explotación del espacio).
- Diversificación es la cantidad de esfuerzo empleado en la búsqueda de regiones distantes del espacio (exploración).

Sin este equilibrio no podríamos por un lado, identificar rápidamente regiones del espacio con soluciones de buena calidad o por otro lado evitar consumir mucho tiempo en regiones del espacio no prometedoras o ya exploradas.

Existe un abanico de distintas metaheurísticas creadas en función de la necesidad del problema a tratar, cada una aborda de diferente manera los siguientes conceptos:

- Seguimiento de trayectoria considerado
- Uso de poblaciones de soluciones
- Uso de memoria
- Fuente de inspiración

En [7] se puede encontrar una posible clasificación de las metaheurísticas:

- Basadas en métodos constructivos: Estos métodos parten de una solución vacía y van añadiendo componentes hasta obtener una solución de calidad. Tenemos como ejemplos el GRASP y la Optimización Basada en Colonias de Hormigas
- Basadas en trayectorias: Estas metaheurísticas parten de una solución inicial y aplicando un algoritmo de búsqueda local, van aplicando cambios a la solución de partida. Si se representa la búsqueda local en una gráfica se vería que sigue una trayectoria en el espacio de búsqueda, de ahí su nombre. Tenemos como ejemplos la Búsqueda Local o la Búsqueda TABU.
- Basadas en poblaciones: El proceso considera múltiples puntos de búsqueda en el espacio que evolucionan en paralelo. Un ejemplo serían los algoritmos genéticos.

En [8] se pueden encontrar la misma clasificación añadiendo un grupo mas, las metaheurísticas de relajación las cuales utilizan relajaciones del modelo original cuya solución facilita la solución del problema original.

Nosotros nos vamos a centrar en una metaheurística que usa tanto métodos constructivos como métodos de trayectorias, hablamos del GRASP, Greedy Randomized Adaptative Search Procedure. Esta metaheurística veremos que tiene dos fases, una constructiva y otra de búsqueda local, la fase constructiva tiene definido su algoritmo salvo extensiones que le podemos aplicar pero en la fase de búsqueda local o también llamada de optimización podemos elegir cuál será nuestro algoritmo.

Veremos entonces cuales son los métodos de búsqueda local más comunes dentro de las variantes del VNS, Variable Neighborhood Search, ya que esta es la metaheurística que más se suele aplicar para la segunda fase del GRASP.

2.2. VNS

Como se define en el capítulo 3 de [9], Variable Neighborhood Search (VNS) está basado en la idea de un cambio sistemático de vecindario, con una fase de descenso para encontrar un óptimo y con una fase de perturbación para salir del correspondiente valle. Originalmente fue diseñado para la solución aproximada de problemas de optimización combinatoria aunque se amplió para abordar programas enteros mixtos, programas no lineales y, recientemente, programas enteros mixtos no lineales. Además, el VNS se ha utilizado como herramienta para la teoría de grafos automatizada o asistida por ordenador. Esto ha permitido descubrir más de 1.500 conjeturas en este campo y demostrar automáticamente más de la mitad de ellas. Las aplicaciones son cada vez más numerosas y se refieren a numerosos ámbitos: teoría de la localización, análisis de conglomerados, enrutamiento de vehículos, diseño de redes, inteligencia artificial, etc.

En esta sección formalizaremos un esquema básico del VNS para después ver sus principales variantes y focalizarnos en la que usaremos para el GRASP.

2.2.1. Esquema básico

Planteamos el problema de optimización determinístico tal que

$$\min\{f(x)|x \in X, X \subset \mathscr{S}\}, \tag{2.1}$$

donde $\mathscr{S}, X, x$, y f son el espacio de solución, el conjunto de las soluciones posibles, una posible solución y una función objetivo de variable real, respectivamente. Si $\mathscr{S}$ es un conjunto finito pero grande habremos definido un problema de optimización combinatoria, mientras que si $\mathscr{S} = \mathbb{R}^n$ tendremos un problema de optimización lineal. Una solución $x^* \in X$ es óptima si

$$f(x^*) \leq f(x), \forall x \in X.$$

Un algoritmo exacto para el problema (2.1), si es que existe, encuentra una solución óptima x^* junto con la prueba de que efectivamente es la óptima, o muestra que no existen soluciones, i.e., $X = \emptyset$, o que la solución no está limitada. Además, en la práctica, el tiempo necesario para resolverlo debe ser finito y no demasiado largo. Para la optimización continua, es razonable tener una tolerancia, es decir, que se detenga cuando se detecte una convergencia suficiente.

Denotemos $\mathscr{N}_k, (k = 1, ..., k_{max})$, como un conjunto finito de estructuras de vecindarios preseleccionados, y $\mathscr{N}_k(x)$ el conjunto de soluciones en el k-esimo vecindario de x. La mayoría de las heurísticas de búsqueda local utilizan sólo un vecindario, i.e., $k_{max} = 1$. A menudo, los sucesivos vecindarios $\mathscr{N}_k$ están anidados y pueden inducirse a partir de una o más estructuras métricas (o cuasimétricas) introducidas en un espacio de soluciones $\mathscr{S}$. Una solución óptima x_{opt} (o mínimo global) es una solución factible en la que se alcanza un mínimo. Llamamos $x' \in X$ a un mínimo local de (2.1) con respecto a $\mathscr{N}_k$ (c.r.a $\mathscr{N}_k$ para abreviar), si no existe una solución $x \in \mathscr{N}_k(x') \subset X$ tal que $f(x) < f(x')$. Las metaheurísticas basadas en procedimientos de búsqueda local intentan continuar la búsqueda por otros medios una vez han alcanzado un mínimo local. El VNS esta basado en tres hechos sencillos:

- Un mínimo local c.r.a una estructura de vecindario no lo es necesariamente para otra.
- Un mínimo global es un mínimo local c.r.a todas las posibles estructuras de vecindario.
- Para muchos problemas, mínimos locales c.r.a una o varias $\mathscr{N}_k$ están relativamente cerca unos de otros.

Esta última observación, que es empírica, implica que un óptimo local proporciona a menudo cierta información sobre el global. Por ejemplo, puede haber varias variables que comparten los mismos valores en ambas soluciones. Dado que estas variables no se pueden identificar de antemano, se debe realizar un estudio organizado de los vecindarios de un óptimo local hasta que se encuentre una solución mejor.

Para resolver (2.1) usando varios vecindarios podemos usar los hechos 1 y 3 para escoger la estrategia de diferentes formas: determinístico, estocástico o la combinación de ambos.

2.2.2. Variantes

Vamos estudiar el funcionamiento de cuatro métodos de VNS, para ello previamente definimos esta función común que todos comparten:

Algoritmo 1: Cambio de vecindario

Input: x, x', k
Output: x, k
1 **if** $f(x') < f(x)$ **then**
2 $\quad x \leftarrow x'$ `// Cambiar de solución`
3 $\quad k \leftarrow 1$ `// Estructura de vecindario inicial`
4 **else**
5 $\quad k \leftarrow k+1$ `// Siguiente estructura de vecindario`

El algoritmo [1] describe la función que mueve la solución y cambia la estructura del vecindario. En la línea 1 compara la actual solución $f(x)$ con el nuevo valor propuesto $f(x')$ obtenido en el k-ésimo vecindario. Si se ha conseguido una mejora se actualiza la solución actual y k vuelve a valer 1 otra vez (línea 2 y 3). De lo contrario pasamos a considerar el siguiente vecindario. La solución actual x, actualizada o no, y el valor de k son devueltos.

Pasamos a describir los cuatro métodos de VNS:

- El método Variable Neighborhood Descent (VND) (Algoritmo [2]) realiza un cambio de vecindario en un sentido determinístico ya que no tiene ningún componente aleatorizado. Su funcionamiento consta de comparar el mínimo local de cada vecindario con la solución actual, si es mejor vuelve a la estructura $k = 1$ de la nueva solución $N_1(x')$, si en el vecindario $\mathcal{N}_{k_{max}}(x)$ esta solución es un mínimo local se detiene el algoritmo.

  La mayoría de las heurísticas de búsqueda local utilizan uno o a veces dos vecindarios para mejorar la solución actual (i.e., $k_{max} \leq 2$). Tenga en cuenta que la solución final debe ser un mínimo local con respecto a todos los vecindarios k_{max} y, por lo tanto, es más probable que se alcance un óptimo global que con una sola estructura.

Algoritmo 2: Variable neighborhood descent

Input: x, k_{max}
Output: x
1 $k \leftarrow 1$
2 **for** $k < k_{max}$ **do**
3 $\quad x' \leftarrow arg\min_{y \in N_K(x)} f(y)$ `// Encuentra el mejor vecino en` $N_K(x)$
4 $\quad x, k \leftarrow$ Cambio de vecindario(x, x', k) `// Cambio de vecindario`

- El método Reduced VNS (RVNS) se obtiene al comparar un punto aleatoriamente seleccionado de $\mathcal{N}_k(x)$ con la solución actual sin ninguna búsqueda de mejora. También se suele elegir una condición de parada, como por ejemplo el tiempo máximo de CPU permitido t_{max}, o el número máximo de iteraciones entre dos mejoras, por lo tanto, RVNS (Algoritmo [3]) utiliza dos parámetros: t_{max} y k_{max}.

Algoritmo 3: Reduced VNS

Input: x, k_{max}, t_{max}
Output: x
1 **for** $t < t_{max}$ **do**
2 $k \leftarrow 1$
3 **for** $k < k_{max}$ **do**
4 $x' \leftarrow \text{Agitar}(x,k)$
5 $x,k \leftarrow$ Cambio de vecindario(x,x',k)
6 $t \leftarrow \text{TiempoCpu}()$

La función agitar selecciona un punto x' aleatoriamente de el k-ésimo vecindario de x, i.e., $x' \in \mathcal{N}_k(x)$. Se muestra en el algoritmo [4] donde se asume que los puntos de $\mathcal{N}_k(x)$ están numerados, $x^1, ..., x^{|\mathcal{N}_k(x)|}$

Algoritmo 4: Agitar

Input: x, k
Output: x'
1 $w \leftarrow [1 + \text{Rand}(0,1) \times |\mathcal{N}_k(x)|]$
2 $x' \leftarrow x^w$

- El método Basic VNS (BVNS) combina cambios deterministicos y estocasticos en el cambio de vecindario. Consiste en elegir una solución inicial x, encontrar una dirección de mejora desde x y mover nuestra solución hacia el mínimo de $f(x)$ en $N(x)$ sobre esa dirección, cuando no hay dirección donde moverse el algoritmo se para, de lo contrario itera. Introducimos su algoritmo en [5].

Algoritmo 5: Basic VNS

Input: x, k_{max}, t_{max}
Output: x
1 **for** $t < t_{max}$ **do**
2 $k \leftarrow 1$
3 **for** $k < k_{max}$ **do**
4 $x' \leftarrow \text{Agitar}(x,k)$
5 **for** $f(x'') \leq f(x')$ **do**
6 $x'' \leftarrow x'$
7 $x' \leftarrow arg\min_{y \in N_K(x'')} f(y)$
8 $x'' \leftarrow x'$ $x,k \leftarrow$ Cambio de vecindario(x,x'',k)
9 $t \leftarrow \text{TiempoCpu}()$

- El método General VNS. Este método sale de contemplar la posibilidad de usar el VND en el paso de la búsqueda local del BVNS (líneas 6-10 [5]). Ahora además de los vecindarios $N_1, ..., N_{k_{max}}$ que serán usados en la función Agitar, tenemos los vecindarios $N_1, ..., N_{l_{max}}$ que pertenecerán a la parte del VND. Vemos su algoritmo en [6]

Una vez vistos los principales métodos VNS, seleccionaremos el VND para usarlo en el método GRASP. Esto es debido a que la fase de constructiva del GRASP ya realiza una exploración exhaustiva al iterar multiples veces y nosotros nos queremos centrar en obtener el mínimo local de las rutas ya construidas, es por eso que descartamos los VNS que contengan la función agitar.

Algoritmo 6: General VNS

Input: $x, l_{max}, k_{max}, t_{max}$
Output: x

```
1 for t < t_max do
2     k ← 1
3     for k < k_max do
4         x' ← Agitar(x, k)
5         x'' ← VND(x', l_max) x, k ← Cambio de vecindario(x, x'', k)
6     t ← TiempoCpu()
```

2.3. GRASP

En el capítulo 6 de [9] lo definen como una metaheurística de multiarranque para problemas de optimización combinatoria, donde cada iteración consiste en dos fases: construcción y búsqueda local. Vemos a continuación como es en detalle y como se aplica al TTDP. En su nombre podemos ver dos características suyas básicas, Greedy Randomized, aleatorizado voraz en inglés, está incluido en su nombre ya que cuando este algoritmo tiene que decidir entre varias posibilidades, toma una corta lista con las elecciones que más puntuación le dan, voraz, y de esta escoge de forma aleatoria la definitiva, aleatorizado

Volviendo a nuestro problema, el procedimiento constructivo básico del TTDP consiste en, partiendo de rutas vacías, seleccionar iteractivamente PDIs para incluirlos en una de las rutas. El método voraz constructivo consiste en seleccionar siempre el mejor PDI e insertarlo en la mejor posición de una ruta según la función objetivo que estipulamos, ya que podemos optar por seleccionar los PDIs más valiosos o por consumir el menor tiempo posible añadiendo el PDI, serían dos funciones objetivo válidas para un método voraz.

En cambio, la fase de construcción GRASP selecciona aleatoriamente uno entre los mejores PDI para incluirlo en una ruta. La fase de mejora consiste en aplicar un procedimiento de mejora a la solución proporcionada por la fase de construcción.

El procedimiento de mejora estándar es una búsqueda local basada en un paso de mejora que se aplica a la solución actual mientras sea posible ese cambio. Se pueden aplicar los pasos o movimientos de mejora habituales en los problemas de rutas que intentan reducir la duración de la ruta. Variable Neighbourhood Descent (VND) es una extensión de una búsqueda local derivada de la conocida metaheurística Variable Neighbourhood Search (VNS). VNS se basa en la idea de aplicar cambios sistemáticos en el tipo de movimientos utilizados en la búsqueda de una solución mejor. Variable Neighbourhood Descent (VND) combina aquellos movimientos que reducen la duración de la ruta con la inserción de nuevos PDIs cuando esta reducción crea espacio para ello.

Tanto la fase constructiva como la de mejora se incluyen en una estrategia de multiarranque para un número de iteraciones. En [10] podemos encontrar el procedimiento general y sus dos fases escritas en pseudocódigo, mostramos el procedimiento general en el algoritmo [7].

Explicamos los parámetros input que el procedimiento GRASP recibe:

- El conjunto **puntos** serían todos los PDIs que pueden formar la ruta.
- El entero **interacciones** es el número de veces se va a pasar por el algoritmo, para decidir su valor tendremos que tener en cuenta factores como el nivel de computación que tenemos, la calidad/fiabilidad de la solución esperada o el tamaño del conjunto de los PDIs.

Algoritmo 7: GRASP

Input: puntos, iteraciones, tamaño, k_{max}
Output: solucion

```
1 for iteracion ≤ iteraciones do
2     RutaGuardada = GRASPconst(puntos, tamaño) // fase de construcción
3     RutaGuardada = VNS(puntos, RutaGuardada, k_max) // fase de optimización
4     if f(RutaGuardada) > f(solucion) then
5         solucion = RutaGuardada // se actualiza la solución
6     k = k + 1
```

- El entero **tamaño** es el tamaño de la Restricted Candidate List (RCL) usada para la parte de construcción, dependiendo del número de PDIs convendrá usar un valor más grande, mayor amplitud de busqueda, o un valor más pequeño que dara mayor voracidad en la búsqueda. De todas formas siempre convendrá probar el algoritmo con un rango de valores para ver con cual alcanza mayor eficiencia.

- Por ultimo el numero **kmax** es el máximo tamaño de los vecindarios para ser usado en el Variable Neighbour Descent para la parte de optimización, su valor será propuesto en base a la cantidad de vecindarios que queramos observar o en base al tiempo de computación del que dispongamos.

Estos serían los inputs a grandes rasgos, en el sentido de que con el conjunto de puntos además de los mismos puntos también nos estamos refiriendo a la información de las distancias entre ellos y las propiedades que tienen cuando tenemos restricciones especiales en el problema.

Otro dato de input que podemos echar en falta es el tiempo máximo por ruta que es crucial para el problema pero puede darse el caso de que no se necesite, por ejemplo si tenemos ventanas de tiempo se puede prescindir ya que estas hacen en cierta medida su función, en definitiva este sería el esquema más sencillo y esencial del GRASP a partir del cual nosotros podemos ir cambiandolo al gusto de nuestras nuevas restricciones al problema.

Como output obtenemos el conjunto de las mejores rutas que hemos creado para todos los días, aunque en la practica tambien seria interesante guardarnos los mejores conjuntos de rutas y no solo uno, ya que en nuestro problema es beneficioso no solo tener la que mejor puntaje haya obtenido sino también algunas que le sigan con puntuaciones muy parecidas para que el viajero pudiera elegir bajo criterios suyos más subjetivos.

Se podría pensar que si esos criterios son conocidos se podrían haber añadido directamente a la formulación del problema pero no es viable añadir cualquier particularidad que se le ocurra al viajero o simplemente no puedes incluirlos, como sería el ejemplo de los eventos meteorológicos que pueden hacer que la ruta con mejor puntuación deje de tener sentido de un momento a otro.

Volviendo al algoritmo este comienza en la línea 1 controlando que no nos pasemos del máximo de iteraciones, en la línea 2 tenemos el proceso de construcción y en la 3 el de optimización. El conjunto de mejores soluciones se irá comparando en la línea 4 bajo la función objetivo del problema y si encontramos mejor solución será actualizada en la línea 5. Ya solo nos queda la línea 6 que suma las iteraciones para el control de estas.

Mostramos el procedimiento de la fase de construcción en el algoritmo [8].

La fase de construcción del GRASP, GRASP_const, recibe el conjunto de PDIs y el tamaño de la RCL. Cada ruta está formada por una sucesión de PDIs que el turista va a visitar. El GRASP empieza

Algoritmo 8: Fase construcción GRASP

Input: $puntos$, tamaño
Output: $ruta$

```
1 ruta = (salida, llegada)                          // creación de la ruta vacía
2 CL = puntos                                       // todos los PDIs forman la CL
3 while CL is not empty do
4     evaluar los puntos de la CL                   // usamos una función de evaluación
5     RCL = { los tamaño mejores puntos de CL}      // construimos la RCL
6     p = choice(RCL)                               // seleccionamos uno aleatoriamente
7     añadir punto p a ruta                         // Añadimos el nuevo punto
8     actualizamos CL          // excluimos los puntos ya no factibles de añadir
```

creando rutas vacías de PDIs para cada día, solo le asignaremos el punto de salida y el de llegada (línea 1). Creamos la lista de candidatos CL con todos los posibles PDIs a visitar (línea 2). Repetiremos una serie de pasos hasta que la CL esté vacía (línea 3). En la línea 4 evaluaremos, según la función objetivo que hemos tomado, el valor de seleccionar un PDI en una determinada posición de una determinada ruta para seleccionar los más valiosos y añadirlos a la RCL (línea 5), cogeremos los primeros valores de tantos como tamaño le hayamos dado a la RCL, por ejemplo un PDI puede estar repetido en la RCL por tener la posibilidad de entrar a una posición/dia u otra. Normalmente se suele ordenar la RCL de forma que los PDIs con mayor valor en la función objetivo tengan más prioridad de ser seleccionador en la línea 6, aunque también se puede evaluar de forma completamente aleatoria. Una vez seleccionado el PDIs con su posición y día determinado se añade a la ruta (línea 7). En la línea 8 evaluamos que PDIs seguirán siendo factibles de añadir tras esta modificación de la ruta, sea porque al añadirlos la ruta sobrepasará el límite de tiempo o por cualquier otra restricción, dando lugar a una CL más reducida.

Los pasos anteriores son repetidos hasta que la CL está vacía, qué quiere decir que no hay ninguna posibilidad de añadir ningún PDI más respetando las restricciones. Si la CL fuera menor que el tamaño de la RCL todos los PDIs pasarian directamente a la RCL, aunque se seguirá evaluando su función objetivo para priorizarlos.

Una vez explicada la primera fase pasamos a entender la fase de optimización.

Los procedimientos habituales de la fase de mejora son las búsquedas locales. Una búsqueda local consiste en aplicar iterativamente un operador de mejora a las rutas proporcionadas por la fase de construcción. Operadores habituales en problemas de rutas son el intercambio de cadenas, la inserción o sustitución y el movimiento 2-opt. El intercambio de cadenas consiste en cambiar una cadena de puntos consecutivos de una posición de una ruta a otra posición de la misma ruta o de otra. Una inserción consiste en insertar nuevos puntos en algunas posiciones de las rutas, aunque en nuestro caso no tendría sentido ya que la fase de construcción termina cuando todos los posibles PDIs han sido añadidos. El operador de sustitución elimina algunos puntos de una ruta e inserta otros nuevos. El movimiento 2-opt consiste en eliminar algunos cruces en las rutas.

La aplicación de cada tipo de operador a una solución factible de un problema de optimización se asocia con una estructura de vecindad en el espacio de soluciones. La vecindad de una solución está formada por las soluciones obtenidas aplicando uno de los operadores en cuestión. La mayoría de las búsquedas locales aplican una estrategia voraz sobre una estructura de vecindad dada del espacio de soluciones. Esta estrategia busca iterativamente la mejor solución en la vecindad de la solución actual hasta que no se pueda encontrar ninguna mejora de este modo. Sin embargo, el principal inconveniente de estos métodos es que quedan atrapados en un óptimo local; es decir, una solución que es la mejor en su vecindario pero no es el óptimo global.

Para la fase de mejora se plantea usar una variante de la metaheurística Variable Neighbourhood

Search (VNS), la Variable Neighbourhood Descent (VND). Esta metaheurística es capaz de utilizar de forma organizada varios tipos de operadores de mejora. El método VND consiste en cambiar el vecindario cada vez que la búsqueda local queda atrapada en un óptimo local con respecto a los vecindarios actuales. El método mejora iterativamente la solución actual utilizando una serie de estructuras de vecindad de soluciones N_k para $k = 1,2,...,k_{max}$, siendo k_{max} el número de estructuras de vecindario. El procedimiento VND se muestra en la figura [9].

Notar que para solucionar el TTDP también es común utilizar solo una metaheurística VNS, para este caso se parte de una solución inicial aleatoria y se le aplica la misma fase de optimización que hemos expuesto, vemos un ejemplo en [11],

Algoritmo 9: Fase optimización VND

Input: $ruta, puntos, k_{max}$
Output: $ruta$

```
1 k = 1                                                    // vecindario inicial
2 while k ≤ k_max do
3     encuentra el mejor cambio en N_k(ruta)               // estrategia voraz
4     if cambio es mejor que ruta then
5         ruta = cambio                  // seleccionamos la ruta como la solución
6         k = 1                        // empezamos con cambios iniciales de nuevo
7     else
8         k = k + 1                              // pasamos a otros tipos de cambios
```

Capítulo 3

Aplicación y Resultados

En el capítulo anterior hemos visto la teoría, hemos visto que tendría que cumplir nuestro algoritmo para que estemos aplicando el GRASP, sin embargo en este capítulo veremos las particularidades que surgen al aplicar toda la teoría anterior para resolver el problema a través del lenguaje de programación Python.

También veremos qué resultados obtenemos con nuestro programa y cómo van variando estos resultados a través de jugar con los diferentes parámetros del programa viendo como se comporta con cada combinación de ellos.

3.1. Aplicación

Desde el principio Python fue elegido para llevar la teoría a la práctica ya que queríamos un lenguaje de programación que nos dejara manejar listas a nuestro gusto y no un software de optimización en el que quizás no pudiéramos aplicar todas las restricciones o particularidades que le queríamos añadir. Ya que el GRASP no goza de una lógica muy compleja a aplicar, en un lenguaje de programación como Python estaremos más cómodos adaptando cualquier particularidad al código.

3.1.1. Generación de datos

El primer código a escribir antes de nada es la generación de los datos, necesitamos generar unos datos que se ajusten a como serian en la realidad:

- La matriz que nos da el tiempo de desplazamiento entre un PDI y otro, se genera a partir de colocar tantos puntos aleatoriamente, como PDIs tengamos, en un plano 100x100 y calcular su distancia euclídea truncando el decimal al entero.
- Las puntuaciones de visitar cada PDI se han generado con números aleatorios enteros entre (1,25).
- El tiempo de visita de cada PDI se ha generado de forma similar a las puntuaciones pero entre (1,50)
- El número de categorías se ha estipulado en 4, es decir hay 4 tipos de PDIs. El número mínimo de PDIs por día para cada categoría se calcula con un aleatorio entre 0 y 1 y para el número máximo el mismo proceso entre 2 y 4.
- Para los horarios de apertura y cierre, la apertura es un aleatorio desde las 8 hasta las 18 mientras que el cierre es un aleatorio entre el horario de apertura más 3, se dejan tres horas de margen, y las 22

Además de estos parámetros que tienen este carácter aleatorio debemos especificar también el número de días, la cantidad de PDIs y el tiempo máximo que queremos fijar por día, si es que lo queremos fijar ya que al haber horarios también harían de límite. Para los valores de estos datos crearemos dos conjuntos de valores.

Añadir también que para estas pruebas no vamos a poner casos en los que se prefija uno o una serie de PDIs a una determinada hora o se establezcan diferentes puntos de salida o llegada para cada día.

Es interesante saber que existe la posibilidad de poder prefijar un PDI ya que el turista puede tener un interés particular obligatorio de visitar un PDI a una hora determinada, sin embargo estas fijaciones no tienen relevancia a la hora de probar el rendimiento del algoritmo.

3.1.2. Fase de construcción

Una vez tenemos los datos de prueba debemos implementar los algoritmos antes mostrados. Como es lógico comencé programando la fase de construcción, más en concreto lo primero fue programar el OP, es decir, nuestro problema con solo un dia y sin restricciones, lo siguiente fue ir cambiando el programa ya existente para hacer la ruta en varios días, el TOP, y por último volví a modificar el programa para que tuviera las dos restricciones especiales que le hemos querido añadir.

La última parte a nivel de programación quizás fue la más costosa ya que mientras que programar el GRASP para el OP y el TOP se basaba en seguir los algoritmos y pseudocódigos ya estudiados previamente, para la parte de las restricciones hay puntos donde se tiene que decidir cómo atacarlo.

Para programar el TOP hay que decidir qué función objetivo utilizar, tanto para la fase de construcción como para la de optimización. Existen varias posibilidades, una sería que la función objetivo se base en el beneficio del PDI que se está añadiendo, la otra es que se base en el tiempo que se pierde al añadir el nuevo PDI a la ruta. Notar que además de modificar la función objetivo también se puede crear una RCL más restringida como se hace para el modelo fuzzy GRASP en [12].

Para comprobar que posibilidad da el mejor resultado para nuestro problema debemos hacer pruebas con ambas posibilidades y observar el rendimiento de cada una, siendo posible también hacer una función objetivo mixta.

A la hora de implementar la restricción de horarios primeramente he tenido que asignar una hora de comienzo a la ruta, mi idea ha sido que al añadir el primer PDI le asignaba como hora de finalización de su visita la hora media entre el cierre y la apertura. La idea de esto es que al añadir un nuevo PDI cuando lo insertemos en la ruta parte de los PDIs tendrán que ser desplazados a una hora más temprana o más tardana, por ello conviene que el primero añadido tenga esa flexibilidad de movimiento en el tiempo.

Al añadir un nuevo PDI a la ruta su hora de comienzo de visita será la hora de fin de visita de su PDI previo más el tiempo de desplazamiento, por lo que todos los siguientes PDIs se habrán desplazado el tiempo que cuesta añadir ese PDI a la ruta, sin embargo, no se pueden superar los límites de cierre de esos PDIs, en caso de que se superen toda la ruta se adelantara el tiempo necesario, si es que los PDIs previos al nuevo no superan su límite de apertura al adelantarles la hora.

Para este problema se han creado variables que muestran cuánto se pueden desplazar los PDIs en el tiempo y asi saber mas fácil si se puede añadir un determinado PDI o no.

Para la restricción de las categorías a la hora de implementarla ha sido mucho más fácil ya que al añadir un nuevo PDI a la ruta solo tenía que comprobar que no superará el máximo pero el inconveniente viene con llegar al mínimo. El caso es que tras varias creaciones de rutas una gran parte de ellas no llegaban al mínimo de PDIs de una categoría por dia, entonces la ruta no cumplia los requisitos y no era válida, esto es comprensible ya que en la construcción no se tiene en cuenta primar las categorías que

tienen mínimos, para ello he implementado una lógica que ayuda a lograr ese valor sin modificar apenas el algoritmo.

La idea es que a partir de la RCL del algoritmo [8] creemos otra lista de candidatos más restringida con solo PDIs que pertenezcan a categorías de las cuales no se ha cumplido su mínimo en el dia en el que va a ser añadido el PDI, si esta lista estuviera vacía se procederá de la forma habitual. De esta forma seguimos eligiendo PDIs únicamente de la RCL pero primamos los que hacen cumplir condiciones de categoría.

Esto por supuesto no hace que ya no tengamos ninguna ruta sin la condición del mínimo cumplida pero ayuda a que muchas más rutas sean válidas.

3.1.3. Fase de optimización

Esta fase ha tenido menos dificultad de programación por todo el código que se ha reciclado de la anterior fase, sin embargo se ha tenido que decidir qué tipo de vecindarios se iban a usar y cómo se iban a explorar.

Para movernos por el vecindario de nuestra ruta construida debemos aplicarle modificaciones y el movimiento elegido es el borrado de uno o varios PDIs de la ruta para posteriormente añadir todos los PDIs que sean posibles.

Vamos a desarrollar el proceso de encontrar la mejor ruta de un vecindario (línea 3, algoritmo [9]) en el siguiente algoritmo [10]:

Algoritmo 10: Búsqueda en el vecindario

Input: *ruta*, *k*, tamaño, *puntos*
Output: *optima*

```
1 mayor_ben = 0
2 for combinacion in combinaciones_borrar do
3     (ruta_aux, puntos) = borrar(combinacion, ruta, puntos) // borramos PDIs de la ruta
4     ruta_aux = GRASPconst(puntos,tamaño, ruta_aux) // Añadimos PDIs a la ruta
5     if f(ruta_aux) > mayor_ben then
6         optima = ruta_aux // se actualiza el mejor
7         mayor_ben = f(ruta_aux) // se actualiza el mejor
```

Por input tenemos, *ruta* es la ruta de la que queremos observar el vecindario, *k* es el tamaño de la estructura del vecindario, tamaño es el tamaño de la RCL que querremos usar cuando tengamos que usar la fase de construcción del GRASP y por último *puntos* son los PDIs que quedan por añadir a la ruta.

Por output tenemos la ruta con mayor puntuación según la función objetivo.

El algoritmo empieza en la línea 1 declarando la variable auxiliar que usaremos para almacenar el beneficio de la mejor ruta calculada. En la línea 2 empieza un for en el que va a recorrer todas las combinaciones de borrar *k* PDIs, es decir, si $k = 1$ recorrerá el for tantas veces como PDIs haya en la ruta y si es $k = 2$ tantas veces como pares de PDIs haya. En la línea 3 se creará *ruta_aux* como resultado de haber borrado la combinación que tocaba a *ruta*, además se añadirán los PDIs borrados de la ruta a la variable *puntos*.

En la línea 4 vamos a usar la fase de construcción del GRASP partiendo de *ruta_aux* en vez de partir de una ruta vacía, la otra particularidad es que vamos a restringir que no se puedan añadir los PDIs borrados en la línea previa, al menos no en la posición y dia donde estaban, ya que podría haber otros PDIs o posiciones diferentes a añadir.

En la línea 5 comparamos si la nueva ruta creada supera a las ya visitadas del vecindario y en las líneas 6 y 7 se actualizan los datos si es la mejor hasta el momento.

De esta forma hemos definido el vecindario a la vez que vemos cómo explorarlo que es la parte más compleja del VND. Al final la estructura del vecindario cambiará en función de la modificación que le hagamos a la ruta, nosotros hemos elegido la de quitar PDIs para añadir otros o en distinta posición pero podríamos haber optado por el hecho de coger dos o más PDIs y cambiar sus posiciones sean de distinto o mismo dia. Por supuesto se pueden combinar estas modificaciones dando lugar a vecindarios más grandes.

Para este caso hemos elegido el borrado frente al cambio de posición ya que cuando se eliminan dos PDIs en la fase de construcción ya se está considerando que uno vaya al sitio del otro, entonces mezclar las dos modificaciones hubiera dado lugar a un algoritmo que pasara más de una vez por los mismos puntos de vecindario.

3.2. Resultados

En esta sección vamos a ver finalmente qué resultados nos puede dar nuestro algoritmo con los diferentes parámetros con los que podemos ir jugando.

Repetiremos las mismas pruebas para dos conjuntos de datos de prueba diferentes los cuales se diferenciarán por su tamaño, es decir, uno tendrá más PDIs, más días y más tiempo que el resto mientras que el otro conjunto tendrá menor número de los tres parámetros nombrados. De esta forma veremos como el algoritmo se comporta en diferentes escenarios

Para ello primero vamos a recopilar los parámetros que debemos ajustar antes de realizar una prueba:

- Iteraciones máximas, acorde con la capacidad de cómputo con la que se han ejecutado los resultados fijamos el número en 400 para el conjunto de datos pequeño y en 200 para el grande, ya que a más PDIs aumenta considerablemente el costo computacional.
- Tamaño máximo de las estructuras de vecindario, vamos a elegir $k_{max} = 2$ ya que con $k_{max} = 1$ se queda mucho espacio de soluciones por observar mientras que con $k_{max} = 3$ se sobrecarga la potencia de cómputo.
- Tamaño de la RCL, probaremos los valores más comunes como son 3, 5, 7 y 10.
- Función objetivo, como hemos dicho antes debemos elegir entre premiar el beneficio del PDI o el coste de tiempo de añadirlo, este será el parámetro más relevante para el estudio del rendimiento.
- Los datos de prueba también jugarán un papel aquí ya que probaremos con diferentes, números de días, números de PDIs y tiempo máximo a consumir.

Finalmente después de fijar dos parámetros iguales para todas las pruebas nos quedan dos parámetros con los que alternar, el tamaño de la RCL que podrá ser 3, 5, 7 o 10 y la función objetivo para la que tenemos tres modelos distintos:

El modelo 1 trata de minimizar el coste de tiempo al añadir un PDI, el valor a minimizar es el tiempo de ir al PDI previo sumando el de ir al posterior restando el tiempo que tienen el PDI previo y el posterior entre sí sumado al tiempo de visita del PDI. En definitiva es todo el tiempo que se le suma a la ruta al insertar el PDI.

El modelo 3 busca maximizar la puntuación de la ruta de forma directa, el valor a maximizar es la ponderación del PDI a añadir.

El modelo 2 es una mezcla de los dos, el valor a minimizar es el mismo valor que en el modelo 1 pero a este le vamos a restar la ponderación del PDI multiplicada por 3. De esta forma hacemos que cuente por igual el tiempo perdido con el beneficio ganado, ya que como se podrá ver en los resultados el promedio del tiempo usado partido del beneficio obtenido en las rutas es 3.

Comenzamos con el caso de datos menor, generamos un conjunto de 16 PDIs, contando el nodo de salida/llegada, a repartir en 2 días con 300 minutos.

Los resultados del algoritmo se muestran en el cuadro [3.1], además para una mejor visualización de los beneficios podemos observar [3.2].

Cuadro 3.1: Resultados conjunto de datos pequeño

Modelo	RCL	Ruta	Tiempo usado	Categorías	Beneficio
1	3	[[0, 3, 13, 9, 12, 1, 0], [0, 11, 4, 10, 7, 0]]	[298, 296]	[[3, 1, 1, 0], [1, 0, 1, 2]]	120
1	5	[[0, 3, 13, 9, 1, 6, 0], [0, 15, 4, 10, 12, 0]]	[295, 298]	[[2, 1, 2, 0], [1, 0, 1, 2]]	134
1	7	[[0, 3, 13, 9, 1, 6, 0], [0, 12, 10, 4, 15, 0]]	[295, 298]	[[2, 1, 2, 0], [1, 0, 1, 2]]	134
1	10	[[0, 11, 4, 12, 1, 9, 0], [0, 15, 13, 3, 6, 0]]	[294, 298]	[[2, 0, 2, 1], [1, 1, 1, 1]]	125
2	3	[[0, 6, 13, 3, 15, 0], [0, 11, 10, 12, 9, 0]]	[281, 293]	[[1, 1, 1, 1], [1, 0, 1, 2]]	**135**
2	5	[[0, 3, 13, 9, 1, 6, 0], [0, 15, 4, 10, 12, 0]]	[295, 298]	[[2, 1, 2, 0], [1, 0, 1, 2]]	134
2	7	[[0, 6, 1, 3, 13, 0], [0, 15, 4, 11, 12, 9, 0]]	[263, 299]	[[2, 1, 1, 0], [1, 0, 2, 2]]	125
2	10	[[0, 11, 4, 10, 7, 0], [0, 12, 13, 6, 0]]	[296, 281]	[[1, 0, 1, 2], [1, 1, 1, 0]]	126
3	3	[[0, 5, 14, 4, 0], [0, 6, 1, 12, 15, 0]]	[291, 293]	[[1, 1, 1, 0], [2, 0, 1, 1]]	117
3	5	[[0, 15, 12, 9, 13, 0], [0, 14, 1, 6, 0]]	[276, 289]	[[1, 1, 1, 1], [2, 0, 1, 0]]	130
3	7	[[0, 14, 8, 0], [0, 6, 1, 12, 13, 0]]	[294, 293]	[[1, 0, 1, 0], [2, 1, 1, 0]]	107
3	10	[[0, 15, 12, 13, 9, 0], [0, 14, 1, 6, 0]]	[293, 289]	[[1, 1, 1, 1], [2, 0, 1, 0]]	130

Cuadro 3.2: Beneficios

	Tamaño RCL			
Función objetivo	3	5	7	10
Modelo 1	120	134	134	125
Modelo 2	**135**	134	125	126
Modelo 3	117	130	107	130

El tiempo de cómputo ha sido aproximadamente de 1 minuto por cada ejemplo. El beneficio es calculado sumando las ponderaciones de los PDIs de la ruta, estas ponderaciones son:

scores = [21 , 4 , 2 , 2 , 6 , 17 , 23 , 16 , 6 , 11 , 20 , 11 , 25 , 25 , 24 , 18]

Sobre el tamaño de la RCL tenemos que el tamaño 5 ha sacado la mejor media con diferencia aunque el mejor valor se haya obtenido con RCL = 3 y el modelo 2. Sobre los modelos, el número 2, el modelo mixto, ha obtenido la mejor actuación mientras que el modelo 3 parece no ser el ideal para este conjunto de datos.

Una desventaja que ha tenido el modelo 3 para estas pruebas es que la mayoría de sus rutas son descartadas por no llegar al mínimo de PDIs de una categoría en un día mientras que para el modelo 1 o el 2 menos de una tercera parte se descartan. Esto es debido a que el modelo 3 genera rutas con beneficios, similares al modelo 1 o 2 pero con menor cantidad de PDIs, algo que se puede esperar ya que un modelo persigue ahorrar tiempo mientras que el otro adquirir los mejores. En el conjunto de datos grande esta desventaja se rebaja.

Nos fijamos en que el tiempo usado nunca puede superar el valor 300 en ninguno de sus días. En cuanto a las categorías, según los datos de prueba, las categorías 1 y 3 tienen un mínimo de un PDI en cada día de la ruta. El valor [[3, 1, 1, 0], [1, 0, 1, 2]] quiere decir que el primer dia, lo que representa [3, 1, 1, 0], tiene 3 PDIs de la primera categoría, 1 de la segunda, 1 de la tercera y ninguno de la cuarta. Podemos ver que ningún ejemplo tiene un 0 en las posiciones 1 y 3, por lo tanto satisfacen la condición.

Una vez hechas las pruebas con el conjunto pequeño vamos con el conjunto grande. Antes teníamos 16 PDIs, 2 días y 300 minutos para cada dia, ahora contamos con 31 PDIs, 3 días y 400 minutos, es justamente el doble de PDIs para el doble de tiempo.

Para este caso en lugar de mostrar la ruta, ya que ahora su tamaño ha aumentado y esta carece de significado, mostraremos su numero de PDIs que contiene, véase el cuadro [3.3] para los resultados y el cuadro [3.4] para el resumen de los beneficios.

Cuadro 3.3: Resultados conjunto mayor

Modelo	RCL	N. PDIs	Tiempo usado	Categorías	Beneficio
1	3	20	[392, 360, 367]	[[2, 2, 1, 2], [2, 1, 1, 2], [2, 1, 1, 3]]	279
1	5	20	[399, 394, 392]	[[2, 1, 1, 3], [2, 2, 1, 3], [2, 0, 2, 1]]	284
1	7	19	[387, 374, 388]	[[2, 1, 1, 2], [2, 1, 1, 1], [1, 2, 2, 3]]	**286**
1	10	20	[385, 367, 397]	[[2, 3, 1, 2], [2, 0, 1, 3], [2, 0, 3, 2]]	283
2	3	20	[384, 373, 391]	[[2, 2, 1, 2], [2, 1, 1, 2], [2, 0, 3, 2]]	273
2	5	20	[384, 320, 378]	[[2, 2, 1, 2], [2, 2, 1, 2], [2, 0, 2, 2]]	279
2	7	20	[371, 396, 390]	[[2, 1, 2, 3], [2, 2, 1, 1], [1, 1, 2, 2]]	282
2	10	19	[389, 380, 395]	[[2, 1, 1, 3], [2, 1, 1, 3], [2, 2, 1, 0]]	283
3	3	18	[383, 397, 399]	[[1, 1, 1, 0], [2, 2, 1, 2], [2, 2, 1, 3]]	280
3	5	18	[398, 391, 390]	[[2, 1, 1, 1], [2, 1, 1, 3], [2, 2, 1, 1]]	275
3	7	18	[397, 388, 382]	[[1, 2, 1, 3], [2, 0, 2, 1], [2, 2, 1, 1]]	279
3	10	18	[397, 395, 387]	[[2, 1, 1, 1], [2, 2, 1, 1], [2, 1, 1, 3]]	284

Cuadro 3.4: Beneficios

	Tamaño RCL			
Función objetivo	3	5	7	10
Modelo 1	279	284	**286**	283
Modelo 2	273	279	282	283
Modelo 3	280	275	279	284

El tiempo de cálculo ha sido aproximadamente de 5 minutos para cada ejemplo. En este conjunto tenemos una mayor homogeneidad en los resultados por lo que no podemos tener una idea con certeza de que parámetros hacen rendir mejor al algoritmo. Sin embargo podemos ver como satisface las condiciones de tiempo y categoría. Tambien se puede apreciar como el conjunto menor tenia la mitad de PDIs con la mitad de tiempo respecto al conjunto de grande y hablando de los beneficios se aprecia la misma proporción.

3.3. Conclusión

Una vez obtenidos los resultados pasamos a hacer una conclusión de los resultados del algoritmo.

Es difícil saber cuánto de cerca nos hemos quedado de la solución óptima, por lo difícil o no viable que es calcular esta por un método exacto, a pesar de ello sabemos que las soluciones calculadas son factibles, es decir, cumplen todas las restricciones impuestas y obtienen un valor bastante alejado de una planificación de ruta carente de una técnica de resolución.

Destacamos tres puntos que podríamos mejorar para un posible mejor rendimiento:

- Para la fase de construcción es interesante mirar algún método que modifica al GRASP como es el fuzzy GRASP, descrito en [12]. Sería interesante ver si este modelo, u otra modificación del GRASP, presenta una mejora notable.
- Para la fase de optimización sería interesante probar el rendimiento con otro método de los que hemos visto, véase el BVNS o el GVNS, este último combina el VND con la función agitar [4], lo que daría una mayor exploración de las soluciones junto con la explotación del vecindario que da el VND.
- Para una mayor adaptabilidad al problema que se plantee sería interesante hacer un análisis más exhaustivo de que modelo y tamaño de la RCL funcionan mejor para diferentes tipos de conjuntos, de tener esa información se podría añadir más iteraciones a la mejor combinación en vez de pasar por todos los modelos.

Otra conclusión que podemos sacar además de las relacionadas con el rendimiento es que hemos generado un modelo muy flexible y personalizable, algo muy importante para un destino de aplicación como es el turismo. Nuestro modelo admite empezar desde una ruta con algún PDI ya fijado, admite cambiar la condición del punto fijado de salida y llegada, admite recalcular rutas una vez ya visitados algunos PDIs, todas estas flexibilidades se han logrado sin tenerlas en cuenta, lo que nos da a entender también que el código es muy flexible para cualquier otra modificación particular de un viajante.

Bibliografía

[1] J.A. MORENO, J. BRITO, A. SANTANA, D. CASTELLANOS, I. GARCÍA Y A. EXPÓSITO, *Smart Recommender for Blue Tourism Routing*, 2020.

[2] HAIDER A. ABDULKARIM Y IBRAHIM F. ALSHAMMARI, *Comparison of Algorithms for Solving Traveling Salesman Problem*, 2015

[3] DAMIANOS GAVALAS, CHARALAMPOS KONSTANTOPOULOS, KONSTANTINOS MASTAKAS Y GRAMMATI PANTZIOU, *A survey on algorithmic approaches for solving tourist trip design problems*, 2014

[4] TUSAN DERYA, IMDAT KARA, PAPATYA SEVGIN BICAKCI Y BARIS KECECI, *New Formulations for the Team Orienteering Problem*, 2016

[5] CYNTHIA PORRAS, BORIS PÉREZ-CAÑEDO, DAVID A. PELTA Y JOSÉ L. VERDEGAY, *A Critical Analysis of a Tourist Trip Design Problem with Time-Dependent Recommendation Factors and Waiting Times*, 2022.

[6] JOSÉ RUIZ-MEZA Y JAIRO R. MONTOYA-TORRES, *A systematic literature review for the tourist trip design problem: Extensions, solution techniques and future research lines*, 2022.

[7] FRANCISCO HERRERA, *Introducción a los Algoritmos Metaheurísticos*, 2009.

[8] BELÉN MELIÁN, JOSÉ A. MORENO PÉREZ Y J. MARCOS MORENO VEGA, *Metaheuristics: A global view*, 2003.

[9] MICHEL GENDREAU Y JEAN-YVES POTVIN, *Handbook of Metaheuristics*, 3rd and 6th caption, 3rd edition, 2019.

[10] JOSÉ A. MORENO PÉREZ Y JULIO BRITO SANTANA, *GRASP metaheuristic for the design of Tourist Trips*, 2022.

[11] YAIZA ALEJANDRA RODRÍGUEZ GARCÍA, *Diseño de Rutas turísticas: itinerarios a pie en Santa Cruz de Tenerife*, Universidad de La Laguna, Trabajo de fin de grado, 2021.

[12] AIRAM EXPÓSITO, SIMONA MANCINI, JULIO BRITO Y JOSÉ A. MORENO, *A fuzzy GRASP for the tourist trip design with clustered POIs*, 2019.

Anexo I: Código

Generación y visualización de datos

```
from random import *

n = 16 # Numero de puntos de interes

POIs = [(randint(0, 99), randint(0, 99)) for i in range(n)]
for i in POIs:
    print(i)


def dibuja(POIs):
    dib = []
    for i in range(25):
        dib.append([])
        for j in range(25):
            dib[i].append('  -')
    for k in range(len(POIs)):
        print(k, ': (', int(POIs[k][0] / 4), ',', int(POIs[k][1] / 4), ')')
        dib[int(POIs[k][0] / 4)][int(POIs[k][1] / 4)] = f'{k:3d}'
    print('DIBUJO:')
    print('    ', end='')
    for i in range(25):
        print(f'{i + 1:3d}', end='')
    print()
    for i in range(25):
        print(f'{i + 1:2d}', end=':')
        for j in range(25):
            print(f'{dib[i][j]:3}', end='')
        print()


dibuja(POIs)

max_score = 25
max_visit_time = 50
scores = []
visit_time = []
for i in range(len(POIs)):
    scores.append(randint(1, max_score))

for i in range(len(POIs)):
    print(scores[i])

print('\n \n')

for i in range(len(POIs)):
    visit_time.append(randint(1, max_visit_time))

for i in range(len(POIs)):
```

```
    print(visit_time[i])

dist = []
for i in range(len(POIs)):
    dist.append([])
    for j in range(len(POIs)):
        dist[i].append(int(((POIs[i][0] - POIs[j][0]) ** 2 +
                            (POIs[i][1] - POIs[j][1]) ** 2) ** 0.5))

for i in range(len(POIs)):
    for j in range(n):
        print(f'{dist[i][j]:4d}', end=' ')
    print()


print(scores)
print(visit_time)
print(dist)

#Creacion de ventanas de tiempo

time_window = []
for i in POIs:
    a = randint(8 , 18)
    b = randint(a+3 , 22)
    time_window.append([a,b])
print(time_window)

#Creacion de categorias
n_categories = 4
category_POI = [randint(0,3) for i in POIs]
print(category_POI)
minmax_categories = [[randint(0,1),randint(2,4)] for i in range(n_categories)]
print(minmax_categories)
```

Datos de prueba generados para el conjunto menor

```

scores = [21, 4, 2, 2, 6, 17, 23, 16, 6, 11, 20, 11, 25, 25, 24, 18]
visit_time = [44, 2, 49, 20, 22, 12, 48, 43, 43, 33, 23, 19, 29, 40, 48, 39]
dist = [[0, 67, 12, 34, 63, 89, 48, 78, 100, 42, 88, 58, 61, 33, 78, 34],
        [67, 0, 61, 36, 58, 28, 28, 38, 62, 26, 53, 52, 31, 33, 37, 55],
        [12, 61, 0, 31, 67, 85, 39, 78, 101, 37, 89, 62, 60, 28, 77, 40],
        [34, 36, 31, 0, 41, 55, 30, 46, 69, 10, 58, 35, 28, 8, 46, 23],
        [63, 58, 67, 41, 0, 57, 68, 32, 43, 45, 32, 6, 26, 49, 33, 28],
        [89, 28, 85, 55, 57, 0, 56, 26, 40, 48, 37, 53, 34, 56, 24, 68],
        [48, 28, 39, 30, 68, 56, 0, 61, 85, 22, 75, 61, 47, 21, 60, 53],
        [78, 38, 78, 46, 32, 26, 61, 0, 25, 43, 15, 30, 17, 51, 1, 50],
        [100, 62, 101, 69, 43, 40, 85, 25, 0, 68, 12, 44, 41, 75, 25, 68],
        [42, 26, 37, 10, 45, 48, 22, 43, 68, 0, 56, 39, 26, 8, 42, 33],
        [88, 53, 89, 58, 32, 37, 75, 15, 12, 56, 0, 33, 30, 64, 16, 57],
        [58, 52, 62, 35, 6, 53, 61, 30, 44, 39, 33, 0, 21, 43, 31, 24],
        [61, 31, 60, 28, 26, 34, 47, 17, 41, 26, 30, 21, 0, 34, 17, 34],
        [33, 33, 28, 8, 49, 56, 21, 51, 75, 8, 64, 43, 34, 0, 50, 31],
        [78, 37, 77, 46, 33, 24, 60, 1, 25, 42, 16, 31, 17, 50, 0, 50],
        [34, 55, 40, 23, 28, 68, 53, 50, 68, 33, 57, 24, 34, 31, 50, 0]]
time_window = [[18, 21], [16, 20], [16, 20], [15, 20], [16, 20], [9, 16],
               [9, 21], [16, 21], [9, 18], [17, 22], [15, 22], [14, 18],
               [14, 22], [17, 20], [12, 18], [16, 21]]
n_categories = 4
category_POI = [0, 0, 1, 0, 2, 1, 2, 0, 2, 2, 3, 3, 0, 1, 0, 3]
```

```
minmax_categories = [[1, 4], [0, 4], [1, 5], [0, 2]]
```

Datos de prueba generados para el conjunto mayor

```
scores = [1, 13, 4, 1, 14, 13, 20, 24, 24, 21, 15, 21, 7, 13, 7, 5, 10, 22,
          19, 13, 6, 20, 11, 23, 21, 18, 1, 1, 15, 2, 17]
visit_time = [10, 9, 26, 29, 10, 22, 9, 6, 46, 27, 26, 49, 9, 8, 15, 12, 48,
              20, 3, 30, 13, 25, 23, 49, 47, 11, 37, 29, 27, 33, 6]
dist = [[0, 33, 69, 22, 60, 51, 28, 90, 76, 9, 67, 27, 48, 53, 82, 71, 75, 117,
        93, 61, 36, 45, 67, 81, 47, 13, 61, 50, 116, 73, 104],
        [33, 0, 49, 14, 47, 42, 23, 56, 48, 37, 56, 27, 21, 24, 63, 45, 44, 84,
        66, 50, 5, 26, 34, 49, 18, 26, 50, 30, 83, 43, 71],
        [69, 49, 0, 61, 16, 24, 72, 55, 20, 77, 21, 75, 28, 63, 102, 14, 29,
        70, 28, 19, 53, 73, 57, 34, 60, 70, 19, 20, 69, 24, 56],
        [22, 14, 61, 0, 56, 49, 12, 70, 62, 23, 65, 15, 34, 31, 63, 58, 59, 97,
        80, 58, 14, 24, 45, 63, 25, 12, 58, 41, 97, 58, 85],
        [60, 47, 16, 56, 0, 9, 68, 68, 34, 69, 9, 71, 30, 66, 106, 28, 43, 85,
        44, 4, 52, 73, 64, 48, 62, 62, 4, 18, 84, 38, 71],
        [51, 42, 24, 49, 9, 0, 61, 72, 41, 60, 16, 64, 29, 64, 104, 34, 48, 92,
        53, 10, 48, 68, 65, 53, 59, 55, 10, 18, 91, 43, 78],
        [28, 23, 72, 12, 68, 61, 0, 73, 71, 24, 77, 4, 44, 29, 54, 68, 66, 101,
        88, 70, 20, 17, 46, 70, 24, 15, 70, 52, 100, 66, 90],
        [90, 56, 55, 70, 68, 72, 73, 0, 35, 93, 75, 77, 46, 46, 69, 40, 25, 28,
        41, 72, 55, 61, 27, 21, 48, 82, 72, 54, 27, 30, 19],
        [76, 48, 20, 62, 34, 41, 71, 35, 0, 83, 41, 75, 28, 55, 91, 6, 11, 50,
        17, 38, 51, 67, 44, 14, 53, 73, 38, 28, 49, 7, 36],
        [9, 37, 77, 23, 69, 60, 24, 93, 83, 0, 76, 23, 54, 52, 78, 78, 81, 121,
        100, 70, 38, 42, 68, 86, 47, 11, 70, 58, 120, 79, 108],
        [67, 56, 21, 65, 9, 16, 77, 75, 41, 76, 0, 80, 40, 76, 116, 35, 50, 91,
        47, 6, 62, 82, 74, 55, 72, 71, 6, 28, 90, 45, 77],
        [27, 27, 75, 15, 71, 64, 4, 77, 75, 23, 80, 0, 48, 32, 55, 72, 70, 105,
        92, 73, 24, 19, 50, 74, 28, 14, 73, 56, 104, 70, 94],
        [48, 21, 28, 34, 30, 29, 44, 46, 28, 54, 40, 48, 0, 36, 76, 24, 27, 71,
        45, 34, 24, 44, 35, 33, 32, 44, 34, 12, 70, 25, 57],
        [53, 24, 63, 31, 66, 64, 29, 46, 55, 52, 76, 32, 36, 0, 40, 54, 46, 74,
          71, 70, 19, 15, 18, 49, 6, 41, 70, 48, 73, 48, 64],
        [82, 63, 102, 63, 106, 104, 54, 69, 91, 78, 116, 55, 76, 40, 0, 92, 80,
        92, 105, 110, 57, 38, 47, 81, 45, 69, 110, 88, 92, 84, 87],
        [71, 45, 14, 58, 28, 34, 68, 40, 6, 78, 35, 72, 24, 54, 92, 0, 15, 57,
        21, 32, 48, 65, 45, 20, 52, 69, 32, 22, 56, 10, 43],
        [75, 44, 29, 59, 43, 48, 66, 25, 11, 81, 50, 70, 27, 46, 80, 15, 0, 44,
        24, 47, 45, 60, 33, 5, 46, 70, 47, 31, 43, 5, 30],
        [117, 84, 70, 97, 85, 92, 101, 28, 50, 121, 91, 105, 71, 74, 92, 57,
        44, 0, 46, 89, 83, 89, 55, 38, 77, 109, 89, 76, 1, 48, 13],
        [93, 66, 28, 80, 44, 53, 88, 41, 17, 100, 47, 92, 45, 71, 105, 21, 24,
        46, 0, 47, 68, 84, 57, 23, 70, 90, 47, 43, 45, 23, 34],
        [61, 50, 19, 58, 4, 10, 70, 72, 38, 70, 6, 73, 34, 70, 110, 32, 47, 89,
        47, 0, 55, 76, 68, 52, 65, 64, 0, 22, 88, 42, 75],
        [36, 5, 53, 14, 52, 48, 20, 55, 51, 38, 62, 24, 24, 19, 57, 48, 45, 83,
        68, 55, 0, 20, 31, 50, 13, 26, 55, 34, 82, 45, 71],
        [45, 26, 73, 24, 73, 68, 17, 61, 67, 42, 82, 19, 44, 15, 38, 65, 60,
        89, 84, 76, 20, 0, 33, 63, 14, 32, 76, 55, 89, 61, 79],
        [67, 34, 57, 45, 64, 65, 46, 27, 44, 68, 74, 50, 35, 18, 47, 45, 33,
        55, 57, 68, 31, 33, 0, 34, 21, 57, 68, 47, 55, 36, 46],
        [81, 49, 34, 63, 48, 53, 70, 21, 14, 86, 55, 74, 33, 49, 81, 20, 5, 38,
        23, 52, 50, 63, 34, 0, 49, 75, 52, 37, 38, 10, 25],
        [47, 18, 60, 25, 62, 59, 24, 48, 53, 47, 72, 28, 32, 6, 45, 52, 46, 77,
        70, 65, 13, 14, 21, 49, 0, 36, 65, 43, 76, 47, 66],
        [13, 26, 70, 12, 62, 55, 15, 82, 73, 11, 71, 14, 44, 41, 69, 69, 70,
        109, 90, 64, 26, 32, 57, 75, 36, 0, 64, 50, 109, 69, 97],
        [61, 50, 19, 58, 4, 10, 70, 72, 38, 70, 6, 73, 34, 70, 110, 32, 47, 89,
```

```
        47, 0, 55, 76, 68, 52, 65, 64, 0, 22, 88, 42, 75],
        [50, 30, 20, 41, 18, 18, 52, 54, 28, 58, 28, 56, 12, 48, 88, 22, 31,
        76, 43, 22, 34, 55, 47, 37, 43, 50, 22, 0, 75, 27, 62],
        [116, 83, 69, 97, 84, 91, 100, 27, 49, 120, 90, 104, 70, 73, 92, 56,
        43, 1, 45, 88, 82, 89, 55, 38, 76, 109, 88, 75, 0, 47, 13],
        [73, 43, 24, 58, 38, 43, 66, 30, 7, 79, 45, 70, 25, 48, 84, 10, 5, 48,
        23, 42, 45, 61, 36, 10, 47, 69, 42, 27, 47, 0, 34],
        [104, 71, 56, 85, 71, 78, 90, 19, 36, 108, 77, 94, 57, 64, 87, 43, 30,
        13, 34, 75, 71, 79, 46, 25, 66, 97, 75, 62, 13, 34, 0]]
time_window = [[15, 19], [9, 21], [18, 22], [15, 21], [17, 21], [8, 20],
               [17, 21], [14, 18], [15, 21], [9, 20], [17, 21], [16, 19],
               [10, 15], [11, 20], [11, 15], [15, 21], [13, 20], [8, 17],
               [11, 20], [17, 20], [10, 16], [15, 18], [18, 22], [9, 17],
               [11, 15], [16, 20], [8, 20], [15, 18], [16, 19], [11, 21],
               [18, 22]]
n_categories = 4
category_POI = [0, 0, 2, 0, 3, 3, 1, 2, 3, 1, 1, 1, 3, 3, 2, 1, 1, 1, 0, 3, 3,
                0, 2, 0, 0, 0, 2, 3, 2, 0, 0]
minmax_categories = [[1, 2], [0, 4], [1, 3], [0, 3]]
```

Funciones

```

# Funcion de adicion de tiempo, calcula para el POI i
# colocarlo en la ruta j en la posicion k de la ruta R
def time_used(i,j,k,R):

    # dist es el tiempo entre  i y j  y visit_time es el tiempo en el POI i
    distance = ( dist[R[j][k]][i] + dist[i][R[j][k+1]]
                 - dist[R[j][k]][R[j][k+1]] + visit_time[i] )
    return distance

# Calculamos el tiempo ganado al quitar un PDI
def time_recovered(j,k,R):
    distance = ( dist[R[j][k-1]][R[j][k]] + dist[R[j][k]][R[j][k+1]]
                 - dist[R[j][k-1]][R[j][k+1]] + visit_time[R[j][k]] )
    return distance

#Funcion añadir PDIs a la R
def add_point(Tused, categories_count, R_time_gap_down,
              R_time_gap_up, R_finish_times, POI, R, removeds):


    #test_rut = test_rut +1
    # success nos dira si se ha podido añadir un PDI, 1 es exito, 0 fracaso
    success = 1

    # Calculamos los costos en tiempo de añadir cada PDI a cada dia
    # a cada posicion en una lista de "tres dimensiones"
    time = [[0 for i in range(ndays)] for r in POI]

    k = 0
    for i in POI:
        for j in range(ndays):
            time[k][j] = [10001 for k in range(len(R[j])-1)]

            for r in range(len(R[j])-1):

                time[k][j][r]=time_used(i,j,r,R)
        k = k + 1

```

```
    # Restringimos que no se puedan añadir los PDIs en las posiciones
    # especificadas en el parametro removed
    if removeds != [-1, -1]:
        for rem in removeds:
            if rem[0] in POI:
               time[POI.index(rem[0])][rem[1]][rem[2] -1] = 10000

    # Buscamos el PDI con menor costo de tiempo y factible de entrar
    candidate_list = []
    for i in range(nCandidatelist):

        MIN=10000
        mins=[]
        for ii in range(len(POI)):
            for j in range(ndays):
                for min_auxa in range(len(R[j]) -1):
                    min_aux = time[ii][j][min_auxa]


                    if min_aux < MIN:

                        # Si es menos costoso que los anteriores se comprueba
                        # que sea factible por horarios primero
                        smallest_gap_up=min(R_time_gap_up[j][min_auxa +1 : ])
                        smallest_gap_down=min(R_time_gap_down[j][:min_auxa +1])
                        dif_up = 0

                        if smallest_gap_up < time_used(POI[ii],j,min_auxa,R)/60:
                            dif_up  = ( time_used(POI[ii],j,min_auxa,R)/60 -
                                       smallest_gap_up )
                        bool_aux = 1

                        if len(R[j]) == 2:
                            bool_aux = 0

                        if ( min_auxa == 0 and R_finish_times[j][1] -
                            (visit_time[R[j][1]] + dist[R[j][1]][POI[ii]] +
                            visit_time[POI[ii]])/60 > time_window[POI[ii]][0]
                            and R_finish_times[j][1] - (visit_time[R[j][1]] +
                            dist[R[j][1]][POI[ii]])/60 <
                             time_window[POI[ii]][1]):

                            bool_aux = 0

                        if ( min_aux == time[ii][j][-1] and
                           R_finish_times[j][min_auxa] +
                           dist[R[j][min_auxa]][POI[ii]]/60 >
                           time_window[POI[ii]][0] and
                           R_finish_times[j][min_auxa] +(dist[R[j][1]][POI[ii]]
                           +visit_time[POI[ii]])/60< time_window[POI[ii]][1] ):

                            bool_aux = 0

                        if (dif_up <= smallest_gap_down and
                           time_window[POI[ii]][0] + dif_up <
                           R_finish_times[j][min_auxa] +
                           dist[R[j][min_auxa]][POI[ii]]/60 and
                           time_window[POI[ii]][1] + dif_up >
                           R_finish_times[j][min_auxa] +
                           (dist[R[j][min_auxa]][POI[ii]]
                           + visit_time[POI[ii]])/60 ):

                            bool_aux = 0
```

```
                        if bool_aux == 0 :

                            # Se comprueba que no se supere el maximo de PDIs
                            # de su categoria en ese dia
                            # y se comprueba el tiempo maximo gastado del dia
                            if (minmax_categories[category_POI[POI[ii]]][1] >
                               categories_count[j][category_POI[POI[ii]]] and
                               min_aux < Tmax[j] - Tused[j]):
                                    # Se añade a la lista como el favorito
                                    MIN = min_aux
                                    mins.append([ii,j,min_auxa])

        # Si esta vacio salimos del bucle de añadir candidatos
        if mins == []:
            break
        minimun_coordinates = mins[-1]

        # Le cambiamos el costo de ser seleccionado para que ya no se
        # seleccione con un numero que acote los valores de costo
        time[minimun_coordinates[0]][minimun_coordinates[1
        ]][minimun_coordinates[2]] = 10000
        # Tomamos sus indices
        min_index = [POI[minimun_coordinates[0]] , minimun_coordinates[1]
                      , minimun_coordinates[2]]

        # Añadimos a la lista de candidatos
        candidate_list.append(min_index)

    # Si no existe ningun posible punto a añadir salimos de la funcion
    if candidate_list == [] :
        success = 0
    else:

        # Para incentivar que cumpla el minimo de PDI por categoria hacemos
        # una prelista con los PDIs que rellenan categorias necesitadas
        candidate_list_categories = []
        for i in candidate_list:
            if ( categories_count[i[1]][category_POI[i[0]]] <
               minmax_categories[category_POI[i[0]]][0]):
                candidate_list_categories.append(i)


        # Si no hace falta rellenar ninguna categoria o no habia tales PDIs
        # se usa la lista clasica y se elige el PDI a añadir
        if candidate_list_categories == []:
            # obtengo el POI a añadir a la ruta aleatoriamente
            winner = candidate_list[randint(0, len(candidate_list)-1)]
        else:
            # obtengo el POI a añadir a la ruta aleatoriamente
            # entre los favorables a categoria
            winner = ( candidate_list_categories[
                      randint(0, len(candidate_list_categories)-1)])



        timeused = time_used(winner[0], winner[1], winner[2],R)

        # Actualizamos lista de tiempo usado
        Tused[winner[1]] = Tused[winner[1]] + timeused

        # Actualizamos lista de contador de PDI por categoria
        categories_count[winner[1]][category_POI[winner[0]]] = ( 1 +
```

```
        categories_count[winner[1]][category_POI[winner[0]]])


        # Actualizamos la lista de cuando se termina de visitar un PDI
        if len(R[winner[1]]) == 2:

            # Si estaba el dia vacio se inserta a la hora media
            # entre apertura y cerradura del PDI
            R_finish_times[winner[1]].insert(1 , ( time_window[winner[0]][1]
            - time_window[winner[0]][0])/2 + time_window[winner[0]][0] +
            (visit_time[winner[0]])/60 )
        else:

            if ( time_used(winner[0],winner[1],winner[2],R)/60 >
                min(R_time_gap_up[winner[1]][winner[2] +1 : ])):
                    dif_up = time_used(winner[0],winner[1],winner[2],R)/60
                    - min(R_time_gap_up[winner[1]][winner[2] +1 : ])
            else :
                    dif_up = 0

            if winner[2] != 0:

                # Si no se va a añadir como el primero, se añade justo despues
                #  de el PDI anterior a su posicion dada, excepto cuando esto
                # hace superar limites superiores de horario, entonces
                #  se adelantan todos los PDIs el tiempo necesario
                R_finish_times[winner[1]].insert(winner[2] +1 ,
                    R_finish_times[winner[1]][winner[2]] +
                    (dist[R[winner[1]][winner[2]]][winner[0]] +
                    visit_time[winner[0]])/60 -dif_up )

            else:

                # Si se añade como el primero, se terminara de visitar a la
                # hora del siguiente menos su tiempo de visita y el tiempo de
                # traslado, es decir, el resto de PDIs no se mueve, el nuevo
                # se engancha
                R_finish_times[winner[1]].insert(1 ,
                    R_finish_times[winner[1]][1]
                    - (dist[R[winner[1]][1]][winner[0]] +
                    visit_time[R[winner[1]][1]])/60)

        # Se actualizan las listas de los horarios de finalizar
        dif = time_used(winner[0],winner[1],winner[2],R)/60
        if winner[2] != 0:

            for i in range(winner[2] +2, len(R_finish_times[winner[1]])-1):
                R_finish_times[winner[1]][i] = ( R_finish_times[winner[1]][i] +
                    time_used(winner[0],winner[1],winner[2],R)/60 - dif_up )

            for i in range(1,winner[2] +1):
                R_finish_times[winner[1]][i] = ( R_finish_times[winner[1]][i]
                                                - dif_up )

        # Añadimos el PDI a la lista de las brechas de tiempo
        # y actualizamos estas
        R_time_gap_up[winner[1]].insert(winner[2] +1 , time_window[winner[0]][
            1] - R_finish_times[winner[1]][winner[2]+1])
        R_time_gap_down[winner[1]].insert(winner[2] +1 ,
            R_finish_times[winner[1]][winner[2] +1] - visit_time[winner[0]]/60
            - time_window[winner[0]][0])

        if winner[2] != 0:
```

```
229
230             for i in range(winner[2] +2, len(R_finish_times[winner[1]])-1):
231                 R_time_gap_up[winner[1]][i] = ( R_time_gap_up[winner[1]][i] -
232                 time_used(winner[0],winner[1],winner[2],R)/60 + dif_up )
233                 R_time_gap_down[winner[1]][i] = ( R_time_gap_down[winner[1]][i]
234                 + time_used(winner[0],winner[1],winner[2],R)/60 - dif_up )
235
236             for i in range(1,winner[2] +1):
237                 R_time_gap_up[winner[1]][i] = ( R_time_gap_up[winner[1]][i] +
238                 dif_up )
239                 R_time_gap_down[winner[1]][i] = ( R_time_gap_down[winner[1]][i]
240                 - dif_up )
241         # Añadimos el PDI a la ruta y lo borramos de los posibles a seleccionar
242         R[winner[1]].insert(winner[2]+1 , winner[0])
243         # saco de la lista de los POIs el POI que introduzco a la ruta
244         POI.remove(winner[0])
245
246     return (Tused, categories_count, R_time_gap_down, R_time_gap_up,
247              R_finish_times, POI, R, success)
248
249
250 # Funcion borrar PDIs de la ruta
251 def remove_point(removed, Tused, categories_count, R_time_gap_down,
252                  R_time_gap_up, R_finish_times, POI, R):
253
254     success = 1
255     # Pasamos el proceso de eliminar por cada PDI
256     for rem in removed:
257
258         point = rem[0]
259         day = rem[1]
260         pos = rem[2]
261
262         # Tiempo recuperado
263         timerecovered = time_recovered(day,pos,R)/60
264
265         # Actualizamos el tiempo usado
266         Tused[day] = Tused[day] - timerecovered
267
268         # Actualizamos la cuenta de las categorias
269         categories_count[day][category_POI[point]] = categories_count[
270             day][category_POI[point]] - 1
271
272         # Eliminamos y actualizamos la lista de las visitas
273         # y las lista de brecha de tiempo
274         R_finish_times[day].pop(pos)
275         R_time_gap_up[day].pop(pos)
276         R_time_gap_down[day].pop(pos)
277
278         if pos != 1 and pos != len(R[day]) - 2:
279
280             for i in range(pos, len(R_finish_times[day]) -1):
281                 R_finish_times[day][i] = (R_finish_times[day][i]
282                                           - timerecovered/2)
283                 R_time_gap_down[day][i] = (R_time_gap_down[day][i]
284                                            - timerecovered/2)
285                 if R_time_gap_down[day][i] < 0:
286                     success = 0
287
288             for i in range(1, pos -1):
289                 R_finish_times[day][i] =R_finish_times[day][i] +timerecovered/2
290                 R_time_gap_up[day][i] =R_time_gap_down[day][i] -timerecovered/2
291                 if R_time_gap_up[day][i] < 0:
```

```
                    success = 0

        # Quitamos el PDI de la ruta
        R[day].pop(pos)

        # Añadimos el PDI a la lista de los que se pueden seleccionar
        POI.insert(1, point)

        # Ajustamos los indices de los siguientes PDIs que se van a eliminar
        # ya que al eliminar uno se corre la posicion de los de delante
        bool_rem = 0
        for rem_aux in removed:

            if bool_rem == 1 and rem_aux[1] == rem[1] and rem[2] < rem_aux[2]:
                removed[removed.index(rem_aux)][2] = removed[removed.index(
                                                        rem_aux)][2] -1

            if rem == rem_aux:
                bool_rem = 1

    return (Tused, categories_count, R_time_gap_down, R_time_gap_up,
             R_finish_times, POI, R, success)
```

Código principal

```
from random import *
import copy

# Numero maximo de iteraciones del ciclo de construccion de iteracion
itmax = 15
# Tamaño de la lista de candidatos
nCandidatelist = 7
# Tamaño maximo de estructura de vecindario
Kmax = 2
# Numero de rutas
ndays = 2
# Tiempo maximo a usar del dia
Tmax = [300 for i in range(ndays)]


nPOI = len(POIs)
benefits_global = 0

for  iteractions in range(itmax):

    # Iniciamos la variables de trabajo

    # Tiempo usado por dia
    Tused = [0 for i in range(ndays)]

    # Ruta de cada dia
    R=[[0,0] for j in range(ndays)]

    # Cantidad de PDIs por categoria por dia
    categories_count = [[0 for i in range(n_categories)] for j in range(ndays)]

    # Hora a la que se termina de visitar un PDI de la ruta,
    # tiene la misma estructura que R se corresponde con ella
    R_finish_times = [[0,24] for i in range(ndays)]

    # Tiempo que se podria retrasar la visita del PDI
```

```
    # hasta que siga cumpliendo el horario
    R_time_gap_up = [[100,100] for i in range(ndays)]

    # Tiempo que se podria adelantar la visita del PDI hasta
    # que siga cumpliendo el horario
    R_time_gap_down = [[100,100] for i in range(ndays)]

    # Lista de PDIs que se pueden añadir a la ruta
    POI=list(range(1,nPOI))

    # FASE DE CONSTRUCCION
    # Añadimos los PDIs
    success = 1
    while success == 1 :

        (Tused, categories_count, R_time_gap_down, R_time_gap_up,
         R_finish_times, POI, R, success) = add_point(Tused, categories_count,
         R_time_gap_down, R_time_gap_up, R_finish_times, POI, R, [-1,-1])

    # Calculamos el beneficio obtenido en la ruta con la fase construcctiva
    benefits = 0
    for i in R:
        for j in range(1,len(i)-1):
            benefits = benefits + scores[i[j]]



    # FASE DE BUSQUEDA LOCAL

    # Calculamos el numero total de PDIs en la ruta
    nRpoints = 0
    for i in R:
        nRpoints = nRpoints + len(i) - 2

    # Reseteamos variables
    k = 1
    benefits_aux = benefits

    # Aplicamos hasta que el tamaño maximo de vecindario
    while k <= Kmax :

        # Aplicamos esta densa logica para calcular todas las
        # combinaciones de que PDIs eliminar de la ruta
        posiciones = [i for i in range(1,k)]
        posiciones.append(k-1)
        while posiciones !=  [i for i in range(nRpoints - k + 1, nRpoints +1)]:
            removed = []
            for i in range(k-1,-1,-1):
                if posiciones[i] + k-1 -i < nRpoints:
                    posiciones[i] = posiciones[i] + 1
                    for ii in range(i+1,k):
                        posiciones[ii] = posiciones[ii-1] + 1
                    break


            for i in posiciones:
                pos_aux = 0
                day_aux = - 1
                while pos_aux < i:
                    day_aux = day_aux + 1
                    pos_aux = pos_aux + len(R[day_aux]) - 2
                removed.append([R[day_aux][len(R[day_aux]) - 2 - pos_aux + i]
                                ,day_aux, len(R[day_aux]) - 2 - pos_aux + i])
```

```
100
101
102             # Una vez tenemos la combinacion de PDIs a borrar
103             # mapeamos las variables de trabajo a auxiliares para no pisar
104             # la ruta original
105             Tused_aux = copy.deepcopy(Tused)
106             R_aux = copy.deepcopy(R)
107             categories_count_aux = copy.deepcopy(categories_count)
108             R_finish_times_aux = copy.deepcopy(R_finish_times)
109             R_time_gap_up_aux = copy.deepcopy(R_time_gap_up)
110             R_time_gap_down_aux = copy.deepcopy(R_time_gap_down)
111             POI_aux = copy.deepcopy(POI)
112
113             # Borramos los PDIs
114             (Tused_aux, categories_count_aux, R_time_gap_down_aux,
115             R_time_gap_up_aux, R_finish_times_aux, POI_aux, R_aux, success
116             ) = remove_point(removed, Tused_aux, categories_count_aux,
117             R_time_gap_down_aux, R_time_gap_up_aux, R_finish_times_aux,
118             POI_aux, R_aux)
119
120             if success == 1:
121                 # Añadimos todos los PDIs posibles
122                 while success == 1:
123                     (Tused_aux, categories_count_aux, R_finish_times_aux,
124                     R_time_gap_up_aux, R_time_gap_down_aux, POI_aux, R_aux,
125                     success) = add_point(Tused_aux, categories_count_aux,
126                     R_finish_times_aux, R_time_gap_up_aux,
127                     R_time_gap_down_aux, POI_aux, R_aux, removed)
128
129                 # Volvemos a pasar la funcion de añadir ahora sin restringir
130                 # que se añadan los borrados
131                 success = 1
132                 while success == 1:
133                     (Tused_aux, categories_count_aux, R_finish_times_aux,
134                     R_time_gap_up_aux, R_time_gap_down_aux, POI_aux, R_aux,
135                     success) = add_point(Tused_aux, categories_count_aux,
136                     R_finish_times_aux ,R_time_gap_up_aux, R_time_gap_down_aux,
137                     POI_aux, R_aux, [-1,-1])
138
139                 # Comprobamos si se ha logrado mayor
140                 # beneficio que en los anteriores
141                 benefits_news = 0
142                 for i in R_aux:
143                     for j in range(1,len(i)-1):
144                         benefits_news = benefits_news + scores[i[j]]
145                 if benefits_news > benefits_aux:
146
147                     # Se guarda como la mejor temporal del vecindario
148                     benefits_aux = benefits_news
149                     Tused_aux_best = copy.deepcopy(Tused_aux)
150                     R_aux_best = copy.deepcopy(R_aux)
151                     categories_count_aux_best = copy.deepcopy(
152                                                     categories_count_aux)
153                     R_finish_times_aux_best = copy.deepcopy(R_finish_times_aux)
154                     R_time_gap_up_aux_best = copy.deepcopy(R_time_gap_up_aux)
155                     R_time_gap_down_aux_best=copy.deepcopy(R_time_gap_down_aux)
156                     POI_aux_best = copy.deepcopy(POI_aux)
157
158         # Una vez observado todo el vecindario si la mejor ruta del vecindario
159         # mejora a la nuestra la seleccionamos y empezamos la busqueda local
160         # de nuevo con k = 1
161         if benefits_aux > benefits:
162             benefits = benefits_aux
```

```
163             Tused = copy.deepcopy(Tused_aux_best)
164             R = copy.deepcopy(R_aux_best)
165             categories_count = copy.deepcopy(categories_count_aux_best)
166             R_finish_times = copy.deepcopy(R_finish_times_aux_best)
167             R_time_gap_up = copy.deepcopy(R_time_gap_up_aux_best)
168             R_time_gap_down = copy.deepcopy(R_time_gap_down_aux_best)
169             POI = copy.deepcopy(POI_aux_best)
170             k = 1
171 
172             # Volvemos a calcular el numero total de PDIs en la ruta
173             nRpoints = 0
174             for i in R:
175                 nRpoints = nRpoints + len(i) - 2
176         else:
177             k = k + 1
178 
179 
180 
181     # Comprobamos si hemos llegado al minimo de PDIs por categoria y dia,
182     # de lo contrario la ruta no contara como valida
183     for i in categories_count:
184         k = 0
185         for j in i:
186             if  j < minmax_categories[k][0]:
187                 R = [[0,0] for j in range(ndays)]
188                 benefits = 0
189             k = k + 1
190 
191 
192     # Comprobamos si hemos obtenido la mejor ruta hasta el momento
193     if benefits > benefits_global:
194         benefits_global = benefits
195         BestR = R
```